A Linear First-Order Functional Intermediate Language for Verified Compilers

Sigurd Schneider, Gert Smolka, Sebastian Hack

Saarland University, Saarbrücken, Germany

To appear without appendix in Proc. of ITP 2015, Nanjing, China, Springer LNAI

We present the linear first-order intermediate language IL for verified compilers. IL is a functional language with calls to a nondeterministic environment. We give IL terms a second, imperative semantic interpretation and obtain a register transfer language. For the imperative interpretation we establish a notion of live variables. Based on live variables, we formulate a decidable property called coherence ensuring that the functional and the imperative interpretation of a term coincide.

We formulate a register assignment algorithm for IL and prove its correctness. The algorithm translates a functional IL program into an equivalent imperative IL program. Correctness follows from the fact that the algorithm reaches a coherent program after consistently renaming local variables. We prove that the maximal number of live variables in the initial program bounds the number of different variables in the final coherent program. The entire development is formalized in Coq.

1 Introduction

We study the intermediate language IL for verified compilers. IL is a linear functional language with calls to a nondeterministic environment.

We are interested in translating IL to a register transfer language. To this end, we give IL terms a second, imperative interpretation called IL/I. IL/I interprets variable binding as assignment, and function application as *goto*, where parameter passing becomes parallel assignment.

For some IL terms the functional interpretation coincides with the imperative interpretation. We call such terms *invariant*. We develop an efficiently decidable property we call *coherence* that is sufficient for invariance. To translate IL to IL/I, translating to the

coherent subset of IL suffices, i.e. the entire translation can be done in the functional setting.

The notion of a live variable is central to the definition of coherence. Liveness analysis is a standard technique in compiler construction to over-approximate the set of variables the evaluation of a program depends on. Coherence is defined relative to the result of a liveness analysis.

```
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)
```

```
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)
```

Figure 1: Program (a) and (b) computing $F(n, m) := n * (n + 1) * \ldots * m$

Inspired by the correspondence between SSA [8] and functional programming [10, 2], we formulate a register assignment algorithm [9] for IL and show that it realizes the translation to IL/I. For example, the algorithm translates program (a) to program (b). Correctness follows from two facts: First, register assignment consistently renames program (a) such that the variable names correspond to program (b). Second, program (b) is coherent, hence let binding and imperative assignment behave equivalently. Parameter passing in IL/I can be eliminated by inserting parallel assignments [9]. In program (b), all parameters i, n can simply be removed, as they constitute self-assignments.

A key property of SSA-based register assignment is that the number of imperative registers required after register assignment is bounded by the maximal number of simultaneously live variables [9], which allows register assignment to be considered separate from spilling. We show that our algorithm provides the same bound on the number of different variable names in the resulting IL/I term.

1.1 Related Work

Correspondences between imperative and functional languages were investigated already by Landin [11]. The correspondence between SSA and functional programming is due to Appel [2] and Kelsey [10] and consists of a translation from SSA programs to functional programs in continuation passing style (CPS) [15, 1]. Chakravarty et al. [6] reformulate SSA-based sparse conditional constant propagation on a functional language in administrative normal form (ANF) [16]. Our intermediate language IL is in ANF, and a sub-language (up to system calls) of the ANF language presented in Chakravarty et al. [6].

Two major compiler verification projects using SSA exist. CompCertSSA [3] integrates SSA-based optimization passes into CompCert [13]. VeLLVM [19, 18] is an ongoing effort

to verify the production compiler LLVM [12]. Both projects use imperative languages with ϕ-functions to enable SSA, and do not consider a functional intermediate language. As of yet, neither of the projects verifies register assignment in the SSA setting. In the non-SSA setting, a register allocation algorithm, which also deals with spilling, has been formally verified [5].

Beringer et al. [4] use a language with a functional and imperative interpretation for proof carrying code. They give a sufficient condition for the two semantics to coincide which they call Grail normal form (GNF). GNF requires functions to be closure converted, i.e. all variables a function body depends on must be parameters.

Chlipala [7] proves correctness for a compiler from Mini-ML to assembly including mutable references, but without system calls. Register assignment uses an interference graph constructed from liveness information. Chlipala restricts functions to take exactly one argument and requires the program to be closure converted prior to register assignment. This means liveness coincides with free variables and values shared or passed between functions reside in an (argument) tuple in the heap: Effectively, register assignment is function local. Chlipala does not prove bounds on the number of different variables used after register assignment and does not investigate the relationship to α-equivalence.

1.2 Contributions and Outline

- We formally define the functional intermediate language IL and its imperative interpretation, IL/I. We establish the notion of live variables via an inductive definition. We identify terms for which both semantic interpretations coincide via the decidable notion of coherence.
- Inspired by SSA-based register allocation, we formulate a register assignment algorithm for IL and prove that it realizes an equivalence preserving transformation to IL/I. We show the size of the maximal live set bounds the number of names after register assignment.
- All results in this paper have formal Coq proofs, and the development is available online (see Section 9). We omit proofs in the paper for space reasons. This version contains an appendix.

The paper is structured as follows: We introduce the languages in Section 2 and Section 3. Program equivalence is defined in Section 4. We define invariance in Section 5, establish a notion of live variables in Section 6, and present coherence in Section 7. Register assignment is treated in Section 8.

2 IL

Values, Variables, and Expressions We assume a set $\mathbb{V}$ of values and a function $\beta : \mathbb{V} \to \{0, 1\}$ that we use to simplify the semantic rule for the conditional. By convention,

$$
\begin{array}{rll}
\eta ::= & e \mid \alpha & \text{extended expression} \\
\mathit{Term} \ni s, t ::= & \mathsf{let}\, x = \eta\, \mathsf{in}\, s & \text{variable binding} \\
\mid & \mathsf{if}\, e\, \mathsf{then}\, s\, \mathsf{else}\, t & \text{conditional} \\
\mid & e & \text{value} \\
\mid & \mathsf{fun}\, f\, \overline{x} = s\, \mathsf{in}\, t & \text{function definition} \\
\mid & f\, \overline{e} & \text{application}
\end{array}
$$

Figure 2: Syntax of IL

v ranges over $\mathbb{V}$. We use the countably-infinite alphabet $\mathcal{V}$ for names x, y, z of values, which we call *variables*.

We assume a type *Exp* of expressions. By convention, e ranges over *Exp*. Expressions are pure, their evaluation is deterministic and may fail, hence expression evaluation is a function $[\![\cdot]\!] : \mathit{Exp} \to (\mathcal{V} \to \mathbb{V}_\perp) \rightharpoonup \mathbb{V}_\perp$. Environments are of type $\mathcal{V} \to \mathbb{V}_\perp$ to track uninitialized variables. We assume a function $\mathrm{fv} : \mathit{Exp} \to \mathit{set}\, \mathcal{V}$ such that for all environments V, V' that agree on $\mathrm{fv}(e)$ we have $[\![e]\!]\, V = [\![e]\!]\, V'$. We lift $[\![\cdot]\!]$ pointwise to lists of expressions in a strict fashion: $[\![\overline{e}]\!]$ yields a list of values if none of the expressions in $\overline{e}$ failed, and $\perp$ otherwise.

Syntax IL is a functional language with a tail-call restriction and system calls. IL syntactically enforces a first-order discipline by using a separate alphabet $\mathcal{F}$ for names f, g of function type, which we call *labels*. IL uses a third alphabet $\mathcal{A}$ for names α which we call *actions*. The term $\mathsf{let}\, x = \alpha\, \mathsf{in}\, \ldots$ is like a system call α that non-deterministically returns a value. The formal development treats system calls with arguments. Their treatment is straightforward and omitted here for the sake of simplicity.

IL allows function definitions, but does not allow mutually recursive definitions. The syntax of IL is given in Figure 2.

Semantics The semantics of IL is given as small-step relation $\longrightarrow$ in Figure 3. Note that the tail-call restriction ensures that no call stack is required. The reduction relation $\longrightarrow$ operates on **configurations** of the form (F, V, s) where s is the IL term to be evaluated. The semantics does not rely on substitution, but uses an environment $V : \mathcal{V} \to \mathbb{V}_\perp$ for variable definitions and a context F for function definitions. Transitions in $\longrightarrow$ are labeled with *events* ϕ. By convention, ψ ranges over events different from τ.

$$\mathcal{E} \ni \phi ::= \tau \mid v = \alpha$$

A **context** is a list of named definitions. A definition in a context may refer to previous definitions and itself. Notationally, we use contexts like functions: If a context F can be decomposed as $F_1; f : a; F_2$ where $f \notin \mathit{dom}\, F_2$, we write $F f$ for a and F^f for $F_1; f : a$.

$$\textsc{Op}\quad \frac{[\![e]\!]\,V = v}{\begin{array}{lllll} & F & \mid V & \mid \mathsf{let}\,x = e\,\mathsf{in}\,s \\ \xrightarrow{\tau} & F & \mid V[x \mapsto v] & \mid s \end{array}}$$

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

$$\textsc{Extern}\quad \frac{v \in \mathbb{V}}{F \mid V \mid \mathsf{let}\,x = \alpha\,\mathsf{in}\,s \xrightarrow{v=\alpha} F \mid V[x \mapsto v] \mid s}$$

$$\textsc{Let}\quad \frac{}{\begin{array}{lllll} & F & \mid V & \mid \mathsf{fun}\,f\,\overline{x} = s\,\mathsf{in}\,t \\ \xrightarrow{\tau} & F; f : (V, \overline{x}, s) & \mid V & \mid t \end{array}}$$

$$\textsc{App}\quad \frac{[\![\overline{e}]\!]\,V = \overline{v} \qquad Ff = (V', \overline{x}, s)}{\begin{array}{lllll} & F & \mid V & \mid f\,\overline{e} \\ \xrightarrow{\tau} & F^f & \mid V'[\overline{x} \mapsto \overline{v}] & \mid s \end{array}}$$

Figure 3: Semantics of IL

Otherwise, $Ff = \bot$. To ease presentation of partial functions, we treat $f : \bot$ as if f was not defined, i.e. $f \notin \mathit{dom}(f : \bot)$. We write $\emptyset$ for the empty context.

A **closure** is a tuple $(V, \overline{x}, s) \in \mathcal{C}$ consisting of an environment V, a parameter list $\overline{x}$, and a function body s. Since a function f in a context $F; f : \ldots; F'$ can refer to function definitions in F (and to itself), the first-order restriction allows the closures to be non-recursive: function closures do not need to close under labels. An application $f\overline{e}$ causes the function context F to rewind to F^f, i.e. up to the definition of f (rule App). In contrast to higher-order formulations, we do not define closures mutually recursively with the values of the language.

A **system call** $\mathsf{let}\,x = \alpha\ \mathsf{in}\,s$ invokes a function α of the system, which is not assumed to be deterministic. This reflects in the rule Extern, which does not restrict the result value of the system call other than requiring that it is a value. The semantic transition records the system call name α and the result value v in the event $v = \alpha$.

IL is **linear** in the sense that the execution of each term either passes control to a strict subterm, or applies a function that never returns. This ensures no run-time stack is required to manage continuations. While, by contrast, uses sequentialization ; to manage a stack of continuations.

3 Imperative Interpretation of IL: IL/I

We are interested in a translation of IL to an imperative language that does not require function closures at run-time. We introduce a second semantic interpretation for IL which we call IL/I to investigate this translation. IL/I is an imperative language, where variable binding is interpreted as imperative assignment. Function application becomes a *goto*, and parameter passing is a parallel assignment to the parameter names. Closures are replaced by blocks $(\overline{x}, s) \in \mathcal{B}$ and blocks do not contain variable environments.

Consequently, a called function can see all previous updates to variables. For example, the following two programs each return 5 in IL/I, but evaluate to 7 in IL:

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

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

To obtain the IL/I small-step relation $\longrightarrow_I$, we replace the rules F-Let and F-App by the following rules:

$$\text{I-Let}\quad \frac{}{\begin{array}{llll} & L & \mid V & \mid \mathsf{fun}\, f\, \overline{x} = s\, \mathsf{in}\, t \\ \xrightarrow{\tau}_I & L; f : (\overline{x}, s) & \mid V & \mid t \end{array}} \qquad \text{I-App}\quad \frac{[\![\overline{e}]\!]\, V = \overline{v} \qquad Lf = (\overline{x}, s)}{\begin{array}{llll} & L & \mid V & \mid f\, \overline{e} \\ \xrightarrow{\tau}_I & L^f & \mid V[\overline{x} \mapsto \overline{v}] & \mid s \end{array}}$$

4 Program Equivalence

To relate programs from different languages, we abstract from a configuration's internal behavior and only consider interactions with the environment (via system calls) and termination behavior. IL's reduction relation forms a labeled transition system (LTS) over configurations.

Definition 1 A *reduction system* (RS) is a tuple $(\Sigma, \mathcal{E}, \longrightarrow, \tau, \mathit{res})$, s.t.

(1) $(\Sigma, \mathcal{E}, \longrightarrow)$ is a LTS

(2) $\tau \in \mathcal{E}$

(3) $\mathit{res} : \Sigma \to \mathbb{V}_\perp$

(4) $\mathit{res}\, \sigma = v \Rightarrow \sigma \longrightarrow$-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

4.1 Partial Traces

We consider two configurations in an IDRS equivalent, if they produce the same partial traces. A partial trace π adheres to the following grammar:

$$\Pi \ni \pi ::= \epsilon \mid v \mid \perp \mid \psi\pi$$

We inductively define the relation $\triangleright \subseteq \Sigma \times \Pi$ such that $\sigma \triangleright \pi$ whenever σ produces the trace π. In the following, we write trace for partial trace.

$$\text{Tr-Tau}\ \frac{\sigma \xrightarrow{\tau} \sigma' \qquad \sigma' \triangleright \pi}{\sigma \triangleright \pi} \qquad \text{Tr-End}\ \frac{}{\sigma \triangleright \epsilon} \qquad \text{Tr-Trm}\ \frac{\sigma \longrightarrow\text{-terminal}}{\sigma \triangleright \mathit{res}\, \sigma} \qquad \text{Tr-Evt}\ \frac{\sigma \xrightarrow{\psi} \sigma' \qquad \sigma' \triangleright \pi}{\sigma \triangleright \psi, \pi}$$

The traces a configuration produces are given as $\mathcal{P}\sigma = \{\pi \mid \sigma \triangleright \pi\}$.

Definition 2 (Trace Equivalence) $\sigma \simeq \sigma' :\iff \mathcal{P}\sigma = \mathcal{P}\sigma'$

Lemma 1 σ silently diverges if and only if $\mathcal{P}\sigma = \{\epsilon\}$.

4.2 Bisimilarity

We give a sound and complete characterization of trace equivalence via bisimilarity. Bisimilarity enables coinduction as proof method for program equivalence, which is more concise than arguing about traces directly. We say a configuration σ is *ready* if the next step is a system call. We write $\sigma_2 \overset{R}{\rightsquigarrow} \sigma_1$ for $\forall \sigma_1', \sigma_1 \xrightarrow{\phi} \sigma_1' \Rightarrow \exists \sigma_2', \sigma_2 \xrightarrow{\phi} \sigma_2' \wedge \sigma_1' \; R \; \sigma_2'$. We write $\sigma \Downarrow w$ (where $w \in \mathbb{V}_\perp$) if $\sigma \longrightarrow^* \sigma'$ such that σ' is $\longrightarrow$-terminal and $res(\sigma') = w$.

Definition 3 (Bisimilarity) Let $(S, \mathcal{E}, \longrightarrow, res, \tau)$ be an IDRS. Bisimilarity $\sim \subseteq S \times S$ is coinductively defined as the greatest relation closed under the following rules:

$$\textsc{Bisim-Silent}\quad \frac{\sigma_1 \longrightarrow^+ \sigma_1' \qquad \sigma_2 \longrightarrow^+ \sigma_2' \qquad \sigma_1' \sim \sigma_2'}{\sigma_1 \sim \sigma_2} \qquad \textsc{Bisim-Term}\quad \frac{\sigma_1 \Downarrow w \qquad \sigma_2 \Downarrow w}{\sigma_1 \sim \sigma_2}$$

$$\textsc{Bisim-Extern}\quad \frac{\sigma_1 \longrightarrow^* \sigma_1' \qquad \sigma_2 \longrightarrow^* \sigma_2' \qquad \sigma_1', \sigma_2' \text{ ready} \qquad \sigma_1' \overset{\sim}{\rightsquigarrow} \sigma_2' \qquad \sigma_2' \overset{\sim}{\rightsquigarrow} \sigma_1'}{\sigma_1 \sim \sigma_2}$$

Bisim-Silent allows to match finitely many steps on both sides, as long as all transitions are silent. This makes sense for IDRS, but would not yield a meaningful definition otherwise. Bisim-Extern ensures that every external transition of σ_1' is matched by the same external transition of σ_2', and vice versa. This ensures that if two programs are in relation, they react to every possible result value of the external call in a bisimilar way. The premises that σ_1', σ_2' are ready is there to simplify case distinctions by ensuring that the next event cannot be τ.

Theorem 1 (Soundness and Completeness) Let $(S, \mathcal{E}, \longrightarrow, res, \tau)$ be an IDRS and $\sigma, \sigma' \in S$. Then: $\sigma \sim \sigma' \iff \sigma \simeq \sigma'$

The semantics of IL and of IL/I each forms an IDRS. We define *res* such that $res(\sigma) = v$ if σ is of the form (F, V, e) and $[\![e]\!]\, V = v$. Otherwise, $res(\sigma) = \perp$. The definitions for IL/I are analogous. To relate configurations IL to IL/I, we form a reduction system on the sum $\Sigma_F + \Sigma_I$ of the configurations and lift $\longrightarrow$ and *res* accordingly. It is easy to see that the resulting reduction system is internally deterministic. If not clear from context, we use an index σ_F, σ_I to indicate which language a configuration belongs to.

5 Invariance

We call a term *invariant* if it has the same traces in both the functional and the imperative interpretation.

Definition 4 (Invariance) A closed program s is invariant if

$$\forall V,\ (\emptyset, V, s)_F \simeq (\emptyset, V, s)_I$$

Invariance is undecidable. We develop a syntactic, efficiently decidable criterion sufficient for invariance, which we call coherence. Coherence simplifies the translation between IL and IL/I.

Coherence is based on the observation that some IL programs do not really depend on information from the closure. Assume $Ff = (V', \overline{x}, s)$ and consider the following IL reduction according to rule App:

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

If V agrees with V' on all variables X that s depends on, then the configuration could have equivalently reduced to $(F^f, V[\overline{x} \mapsto \overline{v}], s)$. This reduction does not require the closure V' and is similar in spirit to the rule I-App. Coherence is a syntactic criterion that ensures V and V' agree on a suitable set X at every function application. We proceed in two steps:

1. Section 6 introduces the notion of live variables, which identifies a set that contains all variables a program depends on.

2. Section 7 gives the inductive definition of coherence and shows that coherent programs are invariant.

6 Liveness

A variable x is *significant* to a program s and a context L, if there is an environment V and a value v such that $(L, V, s)_I \not\simeq (L, V[x \mapsto v], s)_I$. Significance is not decidable, as it is a non-trivial semantic property.

Liveness analysis is a standard technique in compiler construction to over-approximate the set of variables significant to the evaluation of an imperative program. While usual characterizations of live variables rely on data-flow equations [14], we define liveness inductively on the structure of IL's syntax. To the best of our knowledge, such an inductive definition is not in literature. The inductive definition factorizes the correctness aspect from the algorithmic aspect of liveness analysis.

We embed liveness information in the syntax of IL by introducing annotations for function definitions: The term $\mathsf{fun}\, f\, \overline{x} : X = s\, \mathsf{in}\, t$ is annotated with a set of variables X.

6.1 Inductive Definition of the Liveness Judgment

We define inductively the judgment **live**, which characterizes sound results of a liveness analysis.

$$\Lambda \vdash \mathbf{live}\, s : X \quad \text{where} \quad \begin{array}{rcll} \Lambda & : & \mathit{context}\,(\mathit{set}\,\mathcal{V}) & \text{liveness for functions} \\ X & : & \mathit{set}\,\mathcal{V} & \text{live variables} \\ s & : & \mathit{Exp} & \text{expression} \end{array}$$

The predicate $\Lambda \vdash \mathbf{live}\, s : X$ can be read as X *contains all variables significant to* s *in any context satisfying the assumptions* Λ. The context Λ records for every function f a set of variables X that we call the **globals** of f. Assuming $\overline{x}$ are the parameters of f, we will arrange things such that the set $X \cup \overline{x}$ contains all variables significant for the body of f, but never a parameter of f: $X \cap \overline{x} = \emptyset$. Throughout the paper, Λ is always a (partial) mapping from labels to globals, and X denotes a set of variables.

Live-Op
$$\frac{\mathrm{fv}(\eta) \subseteq X \qquad x \in X' \qquad X' \setminus \{x\} \subseteq X \qquad \Lambda \vdash \mathbf{live}\, s : X'}{\Lambda \vdash \mathbf{live}\, \mathsf{let}\, x = \eta\, \mathsf{in}\, s : X}$$

Live-Exp
$$\frac{\mathrm{fv}(e) \subseteq X}{\Lambda \vdash \mathbf{live}\, e : X}$$

Live-App
$$\frac{X_1 \subseteq X \qquad \mathrm{fv}(\overline{e}) \subseteq X}{\Lambda; f : X_1; \Lambda' \vdash \mathbf{live}\, f\, \overline{e} : X}$$

Live-Cond
$$\frac{\mathrm{fv}(e) \subseteq X \qquad \Lambda \vdash \mathbf{live}\, s_1 : X_1 \qquad X_1 \cup X_2 \subseteq X \qquad \Lambda \vdash \mathbf{live}\, s_2 : X_2}{\Lambda \vdash \mathbf{live}\, \mathsf{if}\, e\, \mathsf{then}\, s_1\, \mathsf{else}\, s_2 : X}$$

Live-Fun
$$\frac{\Lambda; f : X_1 \vdash \mathbf{live}\, s_1 : X_1 \cup \overline{x} \qquad X_1 \cap \overline{x} = \emptyset \qquad \Lambda; f : X_1 \vdash \mathbf{live}\, s_2 : X_2 \qquad X_2 \subseteq X}{\Lambda \vdash \mathbf{live}\, \mathsf{fun}\, f\, \overline{x}\, :\, X_1\, = s_1\, \mathsf{in}\, s_2 : X}$$

Figure 4: Liveness: An approximation of the significant variables for IL/I

6.1.1 Description of the Rules.

Live-Op ensures that all variables free in η are live. Every live variable of the continuation s except x must be live at the assignment. We require x to be live in the continuation. Live-Cond ensures that the live variables of a conditional at least contain the free variables of the condition, and the variables live in the consequence and alternative. Live-Exp ensures that for programs consisting of a single expression e at least the free variables of e are live. Live-App ensures that the free variables of every argument are live, and that the globals X_1 of f are live at the call site. Live-Fun records the annotation X_1 as globals for f in Λ, ensures that $X_1 \cup \overline{x}$ is a large enough live set for the function body, and that X_1 does not contain parameters of f. The live variables X_2 of the continuation t must be live at the function definition.

Theorem 2 (Liveness is Decidable) For all Λ, X and annotated s, it is efficiently decidable whether $\Lambda \vdash \mathbf{live}\, s : X$ holds.

The proof of Theorem 2 is constructive and yields an efficient, extractable decision procedure. The decision procedure recursively descends on the program structure, checking the conditions of the appropriate rule in every step.

6.2 Liveness Approximates Significance

We show that the live variables approximate the significant variables. We write $L \models \Lambda$ if a context L satisfies the assumptions Λ, and define:

$$\frac{\textsc{LiveCtx1}\quad L \models \Lambda \qquad X \cap \overline{x} = \emptyset \qquad \Lambda; f : X \vdash \mathbf{live}\, s : X \cup \overline{x}}{L; f : (\overline{x}, s) \models \Lambda; f : X} \qquad \frac{\textsc{LiveCtx2}}{\emptyset \models \emptyset}$$

LiveCtx1 ensures that X does not contain parameters and that $X \cup \overline{x}$ is a large enough live set for the function body s under the context $\Lambda; f : X$.

We can now formally state the soundness of the live predicate. We prove that if $\Lambda \vdash \mathbf{live}\, s : X$, then X contains at least the significant variables of s in every context L that satisfies the assumptions Λ. We write $V =_X V'$ if V and V' agree on X, that is if $\forall x \in X, Vx = V'x$.

Theorem 3 For every program s, if $\Lambda \vdash \mathbf{live}\, s : X$ and $L \models \Lambda$ and $V =_X V'$, then $(L, V, s)_I \simeq (L, V', s)_I$.

7 Coherence

Coherence is a syntactic condition that ensures that a program is invariant. Coherence is defined relative to liveness information $\Lambda \vdash \mathbf{live}\, s : X$.

In the following programs, the set of globals of f is $\{x\}$. The program on the left is not invariant, while the program on the right is coherent.

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

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

In the program on the left in line 3, the value of x is 5 and disagrees with the value of x in the closure of f. In the program on the right, x was not redefined, hence both IL and IL/I will compute 7. We say a function f is *available* as long as none of f's globals were redefined. The inductive definition of coherence ensures only available functions are applied.

7.1 Inductive Predicate

The coherence judgment is of the form $\boxed{\Lambda \vdash \mathbf{coh}\ s}$, where s is an annotated program and Λ is similar to the context in the liveness judgment. We exploit that contexts realize a partial mapping, and maintain the invariant that Λ maps only *available* functions to

their globals, and all other functions to $\bot$. The inductive definition given below ensures that only available functions are applied.

$$\textsc{Coh-Op}\ \frac{\lfloor\Lambda\rfloor_{\mathcal{V}\setminus\{x\}} \vdash \mathbf{coh}\ s}{\Lambda \vdash \mathbf{coh}\ \mathsf{let}\, x = \eta\, \mathsf{in}\, s} \qquad \textsc{Coh-Exp}\ \frac{}{\Lambda \vdash \mathbf{coh}\ e} \qquad \textsc{Coh-App}\ \frac{\Lambda f \neq \bot}{\Lambda \vdash \mathbf{coh}\ f\,\overline{y}}$$

$$\textsc{Coh-Cond}\ \frac{\Lambda \vdash \mathbf{coh}\ s \qquad \Lambda \vdash \mathbf{coh}\ t}{\Lambda \vdash \mathbf{coh}\ \mathsf{if}\, x\, \mathsf{then}\, s\, \mathsf{else}\, t} \qquad \textsc{Coh-Fun}\ \frac{\Lambda; f : X \vdash \mathbf{coh}\ t \qquad \lfloor\Lambda; f : X\rfloor_X \vdash \mathbf{coh}\ s}{\Lambda \vdash \mathbf{coh}\ \mathsf{fun}\, f\, \overline{x} : X = s\, \mathsf{in}\, t}$$

7.1.1 Description of the Rules.

Coh-Op deals with binding a variable x. Every function that has x as a global (i.e. $x \in \Lambda f$) becomes unavailable, and must be removed from Λ. We write $\lfloor\Lambda\rfloor_X$ to remove all definitions from Λ that require more globals than X. Trivally, $\lfloor\Lambda\rfloor_{\mathcal{V}} = \Lambda$. To remove all definitions from Λ that use x as global, we use $\lfloor\Lambda\rfloor_{\mathcal{V}\setminus\{x\}}$.

Formally, the definition of $\lfloor\Lambda\rfloor_X$ exploits the list structure of contexts:

$$\begin{array}{rcll} \lfloor\emptyset\rfloor_X & = & \emptyset & \lfloor\Lambda; f : X'\rfloor_X = \lfloor\Lambda\rfloor_X; f : X' \quad X' \subseteq X \\ \lfloor\Lambda; f : \bot\rfloor_X & = & \lfloor\Lambda\rfloor_X; f : \bot & \lfloor\Lambda; f : X'\rfloor_X = \lfloor\Lambda\rfloor_X; f : \bot \quad X' \not\subseteq X \end{array}$$

Coh-App ensures only available functions can be applied, since Λ maps functions that are not available to $\bot$. Coh-Fun deals with function definitions. When the definition of a function f is encountered, its globals X according to the annotation are recorded in Λ. In the function body s, only functions that require at most X as globals are available, so the context is restricted to $\lfloor\Lambda; f : X\rfloor_X$.

Theorem 4 (Coherence is Decidable) For all Λ and annotated s, it is efficiently decidable whether $\Lambda \vdash \mathbf{coh}\ s$ holds.

7.2 Coherent Programs are Invariant

Given a configuration (F, V, t) such that $Ff = (V', \overline{x}, s)$, the **agreement invariant** describes a correspondence between the values of variables in the function closure V' and the environment V. If the closure of f is available, the closure environment V' agrees with the primary environment V on f's globals X: $V' =_X V$. We write $F, V \models \Lambda$ if $\forall f \in \mathit{dom}\, F \cap \mathit{dom}\, \Lambda,\ V' =_X V$ (where $\Lambda f = X$ and $Ff = (V', \overline{x}, s)$).

Function application continues evaluation with the function body from the closure. Assume $Ff = (V', \overline{x}, s)$ and consider the IL reduction:

$$(F, V, f\,\overline{e}) \longrightarrow (F^f, V'[\overline{x} \mapsto \overline{v}]a, s)$$

If coherence is to be preserved, s must be coherent under suitable assumptions. We say Λ approximates Λ' if whenever Λf is defined, it agrees with Λ' and define $\Lambda \preceq \Lambda' :\Longleftrightarrow \forall f \in \mathit{dom}\, \Lambda,\ \Lambda f = \Lambda' f$. The **context coherence** predicate $\Lambda \vdash \mathbf{coh}\, F$ ensures that all function bodies in closures are coherent. It is defined inductively on the context:

$$\textsc{CohC-Emp}\ \frac{}{\emptyset \vdash \mathbf{coh}\,\emptyset} \qquad \textsc{CohC-Bot}\ \frac{\Lambda \vdash \mathbf{coh}\,F}{\Lambda; f:\bot \vdash \mathbf{coh}\,F; f:b} \qquad \textsc{CohC-Con}\ \frac{\Lambda' \vdash \mathbf{live}\,s : X \cup \overline{x} \quad \Lambda; f:X \preceq \Lambda' \quad \lfloor \Lambda; f:X \rfloor_X \vdash \mathbf{coh}\ s\ \Lambda \vdash \mathbf{coh}\,F}{\Lambda, f:X \vdash \mathbf{coh}\,F; f:(V,\overline{x},s)}$$

CohC-Con encodes two requirements: First, the body of f must be coherent under the context restricted to the globals X of f (cf. Coh-Fun). Second, $X \cup \overline{x}$ must suffice as live variables for the function body s under some assumptions Λ' such that $\Lambda; f:X$ approximates Λ'. Approximation ensures stability under restriction: $\Lambda \vdash \mathbf{coh}\,F \Rightarrow \lfloor\Lambda\rfloor_X \vdash \mathbf{coh}\,F$.

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

Theorem 5 (Coherence implies Invariance) Let $\Lambda \vdash \mathbf{coh}\ s$ and $\Lambda \vdash \mathbf{coh}\,F$ and $\Lambda' \vdash \mathbf{live}\,s : X$ such that $\Lambda \preceq \Lambda'$. Then for all $V =_X V'$ such that $F, V \models \Lambda$, it holds $(F,V,s)_F \simeq (\mathit{strip}\,F, V', s)_I$.

Theorem 5 reduces the problem of translating between IL/I and IL to the problem of establishing coherence. For the translation from IL to IL/I, it suffices to establish coherence while preserving IL semantics. Since SSA and functional programming correspond [10, 2], the translation from IL/I to IL can be seen as SSA construction [8], and the translation from IL to IL/I, which we treat in the next section, as SSA destruction.

8 Translating from IL/F to IL/I via Coherence

The simplest method to establish coherence while preserving IL semantics is α-renaming the program apart. A renamed-apart program (for formal definition see Subsection 11.3) is coherent, since every function is always available. The properties of α-conversion ensure semantic equivalence.

We present an algorithm that establishes coherence and uses no more different names than the maximal number of simultaneously live variables in the program. This algorithm corresponds to the assignment phase of SSA-based register allocation [9]. The algorithm requires a renamed-apart program as input to ensure that every consistent renaming can be expressed as a function from $\mathcal{V} \to \mathcal{V}$. We proceed in two steps:

1. We define the notion of *local injectivity* for a function $\rho : \mathcal{V} \to \mathcal{V}$. We show that renaming with a locally injective ρ yields an α-equivalent and coherent program $\rho\,s$.

2. We give an algorithm *rassign* and show that it constructs a locally injective ρ that uses the minimal number of different names.

We introduce **more liveness annotations** before every term in the syntax, i.e. wherever a term s appeared before, now a term $\langle X\rangle\,s$ appears that annotates s with the set X. From now on, s,t range over such annotated terms. We define the projection $[\langle X\rangle\,s] = X$. The annotation corresponds directly to the live set parameter X of the relation $\Lambda \vdash \mathbf{live}\,s : X$, hence it suffices to write $\Lambda \vdash \mathbf{live}\,s$ for annotated programs.

8.1 Local Injectivity

We define inductively a judgment $\rho \vdash \textbf{\textit{inj}}\ s$ where $\rho : \mathcal{V} \to \mathcal{V}$ and s is an annotated program. We use the following notation for injectivity on X:

$$f \rightarrowtail X \ :\Longleftrightarrow\ \forall x\, y \in X,\ f\, x = f\, y \Longrightarrow x = y$$

The rules defining the judgement are given below and require ρ to be injective on every live set X annotating any subterm:

$$\textsc{Inj-Op}\ \frac{\rho \rightarrowtail X \qquad \rho \vdash \textbf{\textit{inj}}\ s}{\rho \vdash \textbf{\textit{inj}}\ \langle X \rangle\, \mathsf{let}\, x = \eta\, \mathsf{in}\, s} \qquad \textsc{Inj-Val}\ \frac{\rho \rightarrowtail X}{\rho \vdash \textbf{\textit{inj}}\ \langle X \rangle\, e} \qquad \textsc{Inj-App}\ \frac{\rho \rightarrowtail X}{\rho \vdash \textbf{\textit{inj}}\ \langle X \rangle\, f\, \overline{y}}$$

$$\textsc{Inj-Cond}\ \frac{\rho \rightarrowtail X \qquad \rho \vdash \textbf{\textit{inj}}\ s \qquad \rho \vdash \textbf{\textit{inj}}\ t}{\rho \vdash \textbf{\textit{inj}}\ \langle X \rangle\, \mathsf{if}\, x\, \mathsf{then}\, s\, \mathsf{else}\, t} \qquad \textsc{Inj-Fun}\ \frac{\rho \rightarrowtail X \qquad \rho \vdash \textbf{\textit{inj}}\ s \qquad \rho \vdash \textbf{\textit{inj}}\ t}{\rho \vdash \textbf{\textit{inj}}\ \langle X \rangle\, \mathsf{fun}\, f\, \overline{x}\ :\ X_1\ = s\, \mathsf{in}\, t}$$

Let $\mathcal{V}_B(s)$ be the set of variables that occur in a binding position in s, and $\mathrm{fv}(s)$ be the set of free variables of s. For our theorems, several properties are required:

(1) The program must be without unreachable code, i.e. in every subterm $\mathsf{fun}\, f\, \overline{x} = s\, \mathsf{in}\, t$ it must be the case that f is applied in t.

(2) A variable in $\mathcal{V}_B(s)$ must not occur in a set of globals in Λ. We define $\Lambda \subseteq U \ :\Longleftrightarrow\ \forall f \in dom\, \Lambda,\ \Lambda\, f \subseteq U$.

(3) A variable in $\mathcal{V}_B(s)$ must not occur in the annotation $[s]$. We write $s \subseteq U$ if for every subterm t of s it holds that every $x \in [t]$ is either in U or bound at t in s.

For renamed-apart programs, these conditions ensure that the live set X in Inj-Fun always contains the globals X_1 of f (cf. Live-App).

Theorem 6 Let s be a renamed-apart program without unreachable code such that $\Lambda \vdash \mathbf{live}\, s$, $\Lambda \subseteq \mathrm{fv}(s)$ and $s \subseteq \mathrm{fv}(s)$. Then

$$\rho \vdash \textbf{\textit{inj}}\ s \implies \rho\ (\lfloor \Lambda \rfloor_{[s]}) \vdash \mathbf{coh}\ (\rho\, s)$$

Theorem 6 states that the renamed program $\rho\, s$ is coherent under the assumptions $\rho\ (\lfloor \Lambda \rfloor_{[s]})$, i.e. the point-wise image of $\lfloor \Lambda \rfloor_{[s]}$ under ρ.

Renaming with a locally injective renaming produces an α-equivalent program (for formal definition see Subsection 11.2), and hence preserves program equivalence:

Theorem 7 Let s be a renamed-apart program without unreachable code such that $\Lambda \vdash \mathbf{live}\, s$, $\Lambda \subseteq \mathrm{fv}(s)$ and $s \subseteq \mathrm{fv}(s)$. Let $\rho, d : \mathcal{V} \to \mathcal{V}$ such that ρ is the inverse of d on $\mathrm{fv}(s)$. Then $\rho \vdash \textbf{\textit{inj}}\ s \implies \rho, d \vdash \rho\, s \sim_\alpha s$

8.2 A Simple Register Assignment Algorithm

The algorithm rassign is parametrized by a function $\mathit{fresh} : \mathit{set}\,\mathcal{V} \to \mathcal{V}$ of which we require $\mathit{fresh} X \notin X$ for all finite sets of variables X. Based on *fresh*, we define a function $\mathit{freshlist}\,X\,n$ that yields a list of n pairwise-distinct variables such that $(\mathit{freshlist}\,X\,n) \cap X = \emptyset$. The SSA algorithm must process the program in an order compatible with the dominance order to work [9]. In our case it suffices to simply recurse on s as follows:

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

We prove in Theorem 8 that the algorithm is correct for any choice of *fresh* and *freshlist*, as long as they satisfy the specifications above.

Theorem 8 Let s be renamed-apart such that $\Lambda \vdash \mathbf{live}\,s$, $\Lambda \subseteq \mathrm{fv}(s)$ and $s \subseteq \mathrm{fv}(s)$. Let ρ be injective on $[s]$. Then: $\mathit{rassign}\,\rho\,s \vdash \boldsymbol{\mathit{inj}}\ s$.

Our implementation of *fresh* implements the heuristic of simply choosing the smallest unused variable. Theorem 9 shows that for this choice of *fresh*, the largest live set determines the number of required names. We use $\mathcal{S}(k)$ to denote the set of the k smallest variables, and $\mathcal{V}_O(s)$ to denote the set of variables occurring (free or in a binding position) in s.

Theorem 9 Assume $\mathit{fresh}\,X$ yields a variable less or equal to $|X|$. Let s be renamed-apart such that $\Lambda \vdash \mathbf{live}\,s$, $\Lambda \subseteq \mathrm{fv}(s)$ and $s \subseteq \mathrm{fv}(s)$. Let k be the size of the largest set of live variables in s, and $\mathit{rassign}\,\rho\,s = \rho'$. If $\rho(\mathrm{fv}(s)) \subseteq \mathcal{S}(n)$ then $\rho'(\mathcal{V}_O(s)) \subseteq \mathcal{S}(max\{n, k\})$.

We prove a slightly generalized version of Theorem 9 by induction on s.

9 Formal Coq Development

Each theorem and lemma in this paper is proven as part of a larger Coq development, which is available online[1]. The development extracts to a simple compiler that, for instance, produces program (b) when given program (a) from the introduction as input.

The formalization uses De-Bruijn representation for labels, and named representation for variables. Notable differences to the paper presentation concern the treatment of annotations, the technical realization of the definition of liveness, and the inductive generalizations of Theorems 6-9.

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

10 Conclusion

We presented the functional intermediate language IL and developed the notion of coherence, which provides for a canonical and verified translation between functional and imperative programs. We formulated an register assignment algorithm by recursion on the structure of IL that achieves the same bound on the number of required registers as SSA-based register assignment. Coherence allowed us to justify correctness without directly arguing about program semantics by proving that the algorithm α-renames to a coherent program.

11 Appendix

11.1 Table of Variable Names and Types

Variable	**Type**	**comment**
$\mathbb{V}$	set	set of values
β	$\mathbb{V} \to \{0,1\}$	conversion to truth value
v	$\mathbb{V}$	value
Exp	set	set of expressions
$\mathcal{V}$	set	set of variables
e	*Exp*	expression
x, y, z	$\mathcal{V}$	variables
$\mathcal{F}$	set	set of lables
f, g	$\mathcal{F}$	labels
$\mathcal{A}$	set	set of actions
η	$\mathit{Exp} + \mathcal{A}$	extended expression
α	$\mathcal{A}$	action
Term	set	set of terms
s, t	*Term*	terms
V	$\mathcal{V} \to \mathbb{V}_\perp$	environment
$\mathcal{C}$	set	set of closures
F	context of $\mathcal{C}$	
$\mathcal{E}$	set	set of events
ϕ	$\mathcal{E}$	event
τ	$\mathcal{E}$	silent event
$\mathcal{B}$	set	set of blocks
L	context of $\mathcal{B}$	
Σ	set	set of states (LTS)
σ	Σ	state, configuration
Π	set	set of partial traces
π	Π	partial trace
ϵ	Π	empty trace

11.2 α-Equivalence

We formalize a generalization of alpha equivalence as an inductively defined judgment $\rho, d \vdash s \sim_\alpha t$ where $\rho, d : \mathcal{V} \to \mathcal{V}$ and s, t are terms. The mapping ρ describes how the free variables of s map to free variables of t, and d describes how the free variables of t map to free variables of s. If $\rho, d \vdash s \sim_\alpha t$ holds, then d is the inverse of ρ on $\mathrm{fv}(s)$, i.e.

$$\forall x \in \mathrm{fv}(s),\ d(\rho\, x) = x$$

Symmetrically, ρ is the inverse of d on $\mathrm{fv}(t)$.

The formalization assumes a similar judgment $\rho, d \vdash_{Exp} e \sim_\alpha e'$ for α-equivalence of expressions. The variable case of judgment for expressions explains how ρ and d are used:

$$\textsc{Alpha-Var}\ \frac{\rho x = y \qquad d y = x}{\rho, d \vdash_{Exp} x \sim_\alpha y}$$

Alpha-Var ensures that ρ maps x to y and d maps y to x.

The other rules of the expression judgment are structurally recursive and we omit them.

$$\textsc{Alpha-Op}\ \frac{\rho, d \vdash_{Exp} \eta \sim_\alpha \eta' \qquad \rho[x \mapsto x'], d[x' \mapsto x] \vdash s \sim_\alpha s'}{\rho, d \vdash \mathsf{let}\, x = \eta\, \mathsf{in}\, s \sim_\alpha \mathsf{let}\, x' = \eta'\, \mathsf{in}\, s'} \qquad \textsc{Alpha-Val}\ \frac{\rho, d \vdash_{Exp} e \sim_\alpha e'}{\rho, d \vdash e \sim_\alpha e}$$

$$\textsc{Alpha-App}\ \frac{\forall i,\ \rho, d \vdash_{Exp} e_i \sim_\alpha e_i'}{\rho, d \vdash f\, \overline{e} \sim_\alpha f\, \overline{e'}} \qquad \textsc{Alpha-Cond}\ \frac{\rho, d \vdash_{Exp} e \sim_\alpha e' \qquad \rho, d \vdash s \sim_\alpha s' \qquad \rho, d \vdash t \sim_\alpha t'}{\rho, d \vdash \mathsf{if}\, e\, \mathsf{then}\, s\, \mathsf{else}\, t \sim_\alpha \mathsf{if}\, e'\, \mathsf{then}\, s'\, \mathsf{else}\, t'}$$

$$\textsc{Alpha-Fun}\ \frac{\rho[\overline{x} \mapsto \overline{x'}], d[\overline{x'} \mapsto \overline{x}] \vdash s \sim_\alpha s' \qquad \rho, d \vdash t \sim_\alpha t' \qquad |\overline{x}| = |\overline{x'}|}{\rho, d \vdash \mathsf{fun}\, f\, \overline{x} = s\, \mathsf{in}\, t \sim_\alpha \mathsf{fun}\, f\, \overline{x'} = s'\, \mathsf{in}\, t'}$$

Figure 5: Inductive judgment generalizing α-equivalence

The relation has several pleasant properties.

Lemma 2 (Reflexivity) $id, id \vdash s \sim_\alpha s$

Lemma 3 (Symmetry) $\rho, d \vdash s \sim_\alpha s' \Rightarrow d, \rho \vdash s' \sim_\alpha s$

Lemma 4 (Transitivity) $\rho_1, d_1 \vdash s \sim_\alpha s' \Rightarrow \rho_2, d_2 \vdash s' \sim_\alpha s'' \Rightarrow \rho_1 \circ \rho_2, d_2 \circ d_1 \vdash s \sim_\alpha s'$

We validate our definition and prove soundness with respect to trace equivalence $\simeq$. We define

$$V =_{\rho,d} V' \;:\Longleftrightarrow\; \forall xy, \rho x = y \Rightarrow dy = x \Rightarrow Vx = V'y$$

We relate two closures in the following way:

$$\begin{aligned}&(V, \overline{x}, s) =_\alpha (V', \overline{x'}, s')\\ :\Longleftrightarrow\;& |\overline{x}| = |\overline{x'}| \;\wedge \exists \rho\, d,\; V =_{\rho,d} V' \wedge \rho[\overline{x} \mapsto \overline{x'}], d[\overline{x'} \mapsto \overline{x}] \vdash s \sim_\alpha s'\end{aligned}$$

We then lift $=_\alpha$ point-wise to contexts of the same length.

Theorem 10 If $F =_\alpha F'$ and $V =_{\rho,d} V'$ then $(F, V, s) \simeq (F', V', s')$.

In the formal development we have an additional formalization of IL which uses De-Bruijn representation also for variables (and not just for labels). We give a translation from the named IL to De-Bruijn IL, and prove this translation correct with respect to trace equivalence. We then show that terms that are α-equivalent by our inductive definition translate to identical terms in De-Bruijn representation.

11.3 Definition of Renamed Apart

A program is renamed apart, if every variable x occurring in a binding position does not occur free and x is different from every variable occurring in a different binding position. We formulate an inductive predicate $X \vdash s\,\mathbf{apart}\,X'$ that ensures this property. The predicate maintains the invariant that all free variables of s are in X, and that X' contains exactly the variables occurring in binding positions in s.

$$\textsc{Apart-Op}\ \frac{\mathrm{fv}(e) \subseteq X \qquad X \cup \{x\} \vdash s\,\mathbf{apart}\,X'}{X \vdash \mathsf{let}\, x' = \eta'\, \mathsf{in}\, s'\,\mathbf{apart}\, X' \cup \{x\}} \qquad \textsc{Apart-Val}\ \frac{\mathrm{fv}(e) \subseteq X}{X \vdash e\,\mathbf{apart}\,\emptyset}$$

$$\textsc{Apart-App}\ \frac{\mathrm{fv}(\overline{e}) \subseteq X}{X \vdash f\,\overline{e}\,\mathbf{apart}\,\emptyset} \qquad \textsc{Apart-Cond}\ \frac{\mathrm{fv}(e) \subseteq X \qquad X \vdash s\,\mathbf{apart}\,X_s \qquad X_s \cap X_t = \emptyset \qquad X \vdash t\,\mathbf{apart}\,X_t}{X \vdash \mathsf{if}\, e\, \mathsf{then}\, s\, \mathsf{else}\, t\,\mathbf{apart}\, X_s \cup X_t}$$

$$\textsc{Apart-Fun}\ \frac{X \vdash t\,\mathbf{apart}\,X_t \qquad X \cup \overline{x} \vdash s\,\mathbf{apart}\,X_s \qquad \mathrm{unique}\,\overline{x} \qquad \overline{x} \cap X = \emptyset \qquad (X_s \cup \overline{x}) \cap X_t = \emptyset}{X \vdash \mathsf{fun}\, f\, \overline{x} = s\, \mathsf{in}\, t\,\mathbf{apart}\, X_s \cup X_t \cup \overline{x}}$$

Figure 6: Inductive definition of renamed apart

Lemma 5 (Disjoint) If $X \vdash s\,\mathbf{apart}\,X'$ then $X \cap X' = \emptyset$.

Lemma 6 (Relation to free and bound variables) If $X \vdash s\,\mathbf{apart}\,X'$ and then $\mathrm{fv}(s) \subseteq X$ and $X' = \mathcal{V}_B(s)$.

11.4 A Procedure to Rename Apart

We define the procedure

$$\mathit{apart} : (\mathcal{V} \to \mathcal{V}) \to (\mathit{set}\,\mathcal{V}) \to \mathit{Exp} \to (\mathit{set}\,\mathcal{V}) \times \mathit{Exp}$$

such that $\mathit{apart}\,\rho\,X\,s = (X, s')$ ensures s' is renamed apart and α-equivalent to s. X' contains the newly chosen variables now occurring in binding positions in s'. Theorem 11 and Theorem 12 make these claims precise.

$$\begin{array}{lcl}
\mathit{apart}\,\rho\,X\,(\mathsf{let}\,x = \eta\,\mathsf{in}\,s) & = & (X' \cup \{y\}, \mathsf{let}\,y = \rho\,\eta\,\mathsf{in}\,s') \\
\quad \text{where } (X', s') = \mathit{apart}\,(\rho[x \mapsto y])\,(X \cup \{y\})\,s & & \\
\quad \text{where } y = \mathit{fresh}\,X & & \\
\mathit{apart}\,\rho\,X\,(\mathsf{if}\,e\,\mathsf{then}\,s\,\mathsf{else}\,t) & = & (X_s \cup X_t, \mathsf{if}\,(\rho\,e)\,\mathsf{then}\,s'\,\mathsf{else}\,t') \\
\quad \text{where } (X_s, s') = \mathit{apart}\,\rho\,X\,s & & \\
\quad \text{where } (X_t, t') = \mathit{apart}\,\rho\,(X \cup X_s)\,t & & \\
\mathit{apart}\,\rho\,X\,e & = & (\emptyset, \rho e) \\
\mathit{apart}\,\rho\,X\,(f\,\overline{e}) & = & (\emptyset, f\,(\rho\,\overline{e})) \\
\mathit{apart}\,\rho\,X\,(\mathsf{fun}\,f\,\overline{x} = s\,\mathsf{in}\,t) & = & (X_s \cup X_t \cup \overline{y}, \mathsf{fun}\,f\,\overline{y} = s'\,\mathsf{in}\,t') \\
\quad \text{where } \overline{y} = \mathit{freshlist}\,X\,|\overline{x}| & & \\
\quad \text{where } (X_s, s') = \mathit{apart}\,(\rho[\overline{x} \mapsto \overline{y}])\,(X \cup \overline{y})\,s & & \\
\quad \text{where } (X_t, t') = \mathit{apart}\,\rho\,(X \cup X_s \cup \overline{y})\,t & &
\end{array}$$

Theorem 11 (*apart* renames apart) Let s be a program such that $\rho(\mathrm{fv}(s)) \subseteq X$ and $\mathit{apart}\,\rho\,X\,s = (X', s')$. Then: $X \vdash s'$ **apart** X'.

Theorem 12 (Renaming apart respects α-conversion) Let s be a program such that $\rho(\mathrm{fv}(s)) \subseteq X$ and $\mathit{apart}\,\rho\,X\,s = (X', s')$ and let d be inverse to ρ on $\mathrm{fv}(s)$. Then $\rho, d \vdash s \sim_\alpha s'$.

11.5 Joining the Parts

This section describes how the theorems proven in this paper fit together in a compiler. Assume that the compiler uses IL as an intermediate language, and now wants to produce code for an IL program s. The compiler procedes as follows:

1. Rename s_1 apart, obtaining an α-equivalent program s_2 (Theorem 12).
2. Run the algorithm *rassign* on s_2 to obtain a register assignment ρ. Theorem Theorem 8 ensures ρ is locally injective.
3. Rename s_2 accoding to ρ and obtain s_3, which is α-equivalent (Theorem 7) and coherent (Theorem 6) because ρ is locally injective.
4. Theorem Theorem 5 ensures that s_3 can be seen equivalently as an IL/I program, hence the functional program s_1 has been translated to an imperative program s_3.

References

1. A. W. Appel. *Compiling with Continuations*. Cambridge, England: Cambridge University Press, 1992.
2. A. W. Appel. "SSA is Functional Programming". In: *SIGPLAN Not.* 33.4 (1998).
3. G. Barthe, D. Demange, and D. Pichardie. "A Formally Verified SSA-Based Middle-End - Static Single Assignment Meets CompCert". In: *ESOP*. 2012.
4. L. Beringer, K. MacKenzie, and I. Stark. "Grail: a Functional Form for Imperative Mobile Code". In: *ENTCS* 85.1 (2003).
5. S. Blazy, B. Robillard, and A. W. Appel. "Formal Verification of Coalescing Graph-Coloring Register Allocation". In: *ESOP*. 2010.
6. M. M. T. Chakravarty, G. Keller, and P. Zadarnowski. "A Functional Perspective on SSA Optimisation Algorithms". In: *ENTCS* 82.2 (2003).
7. A. Chlipala. "A verified compiler for an impure functional language". In: *POPL*. 2010.
8. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". In: *TOPLAS* 13.4 (1991).
9. S. Hack, D. Grund, and G. Goos. "Register Allocation for Programs in SSA-Form". In: *CC*. 2006.
10. R. A. Kelsey. "A correspondence between continuation passing style and static single assignment form". In: *SIGPLAN Not.* 30 (3 1995).
11. P. J. Landin. "Correspondence between ALGOL 60 and Church's Lambda-notation: part I". In: *CACM* 8.2 (1965).
12. C. Lattner and V. S. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *CGO*. 2004.
13. X. Leroy. "Formal Verification of a Realistic Compiler". In: *CACM* 52.7 (2009).
14. T. Nipkow and G. Klein. *Concrete Semantics: With Isabelle/HOL*. Springer Publishing Company, Incorporated, 2014.
15. J. C. Reynolds. "The Discoveries of Continuations". In: *LSC* 6.3-4 (1993).
16. A. Sabry and M. Felleisen. "Reasoning about Programs in Continuation-Passing Style". In: *LSC* 6.3-4 (1993).
17. S. Schneider, G. Smolka, and S. Hack. "A First-Order Functional Intermediate Language for Verified Compilers". In: *CoRR* abs/1503.08665 (2015).
18. J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. "Formal Verification of SSA-based Optimizations for LLVM". In: *PLDI*. 2013.
19. J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. "Formalizing LLVM Intermediate Representation for Verified Program Transformations". In: *POPL*. 2012.