Functional elimination of Φ-instructions

Lennart Beringer
Lehrstuhl für Theoretische Informatik
Ludwig-Maximilians-Universität München
Oettingenstrasse 67, 80538 München, Germany
beringer@tcs.ifi.lmu.de

February 13, 2006

Abstract

We present a functional analogue of the elimination of Φ-instructions from Static Single Assignment (SSA) code. Extending earlier work on the relationship between SSA and functional languages we show that transformations from A-normal form (ANF) into a more restrictive form called GNF require the same compensating instructions to be inserted as are commonly inserted during the translation from SSA to machine code. Lifting the translation from the syntactic level to the type level, we introduce type systems that mediate the transition from ANF code into correctly register-allocated machine code and allow code optimisations and transformations to be performed in a typed functional setting.

1 Introduction

The Static Single Assignment (SSA) form [10] is a popular imperative representation of intermediate code, and several program analysis tasks have been shown to benefit from the usage of SSA [16, 18, 25]. In order to maintain the defining property which requires each variable to have a single point of definition, Φ-instructions are introduced which merge the content of variables at the beginning of basic blocks. During the translation from SSA to machine code, Φ-instructions are replaced by register moves in the control flow predecessors. This insertion of compensation code needs to respect the concurrent interpretation of Φ-instructions in a basic block, even if applied to the outcome of intermediate program optimisations that destroy some of the implicit structure of SSA [8].

Appel and Kelsey observed a close correspondence between SSA and restricted forms of functional programming languages [4, 15]. This correspondence is characterised by (1) the isomorphism between mutually tail-recursive, first-order functions and labelled basic blocks, (2) the fact that each formal parameter of such a function f amounts to one Φ-instruction at the beginning of the basic block labelled f, and (3) the correspondence between syntactic (nested) scope and the notion of dominance that governs where Φ-instructions are placed. Indeed, it is possible to define a language for which a functional (call-by-value) semantics with scope-directed static binding coincides with an imperative semantics with interpreted Φ-functions. As was demonstrated by Chakravarty et al. [9], the correspondence may also be used to express intermediate program analysis operating on SSA using concepts and terminology from functional languages.

Since the translation into machine code destroys the structure on which the correspondence rests, optimisations that cannot be performed at SSA level (coalescing of variables, register allocation, ...) cannot be directly modelled as the counterparts of appropriate functional manipulations. Indeed, the elimination of Φ-instructions potentially introduces additional instructions, variables and basic blocks [8, 27].

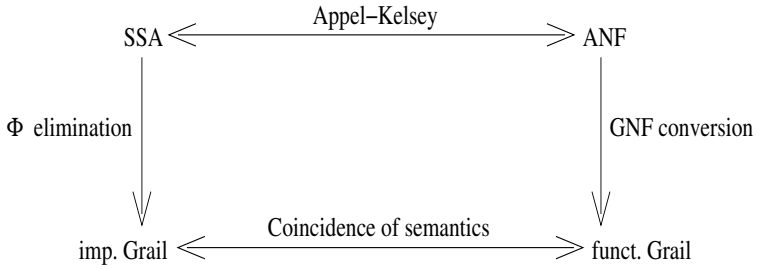


Figure 1: Syntactic Grail conversion and the elimination of Φ-instructions

In previous work [7], we introduced a syntactic discipline on functional code which recovers the correspondence in the absence of Φ-functions. We presented a language (*Grail*) that does not contain Φ-instructions and may be given coinciding functional (call-by-value) and imperative semantics, both of which are defined in an entirely standard way. Moreover, (1) functional (let-bound) variables are in bijection with imperative variables (registers), (2) a free occurrence of a variable corresponds to imperative liveness, and (3) each function call amounts to a single (jump) instruction. In particular, "register shuffling" is explicit, and is performed in compensation instructions that precede the function call. Again, the correspondence could be used for relating program analysis frameworks: we showed that a low-level analysis for detecting when a register content is accessed exactly once could be formalised either imperatively or functionally. The solutions to the appropriate dataflow equations correspond bijectively to the derivations in a certain type system.

In this paper, we show that the correspondences at the language levels extend to the translations between the levels (Figure 1). Starting from A-normal form (ANF, [14]), we define a sequence of transformation steps which yields code in Grail normal form (abbreviated GNF), a restricted form of ANF which embodies the essentials of Grail's syntactic restrictions. In order to demonstrate that the conversion corresponds to the elimination of Φ-instructions, we then consider the effect of each individual step on SSA. We also show that the concurrent interpretation of Φ-instructions is respected. The transformation does not require programs to be in *edge-split* form, but involves a weaker manipulation called *branch normalisation* that is performed as the last step of the transformation. Thus, function identifiers coincide with basic block labels before and after the application of our algorithm.

As a syntactic code representation, GNF violates functional abstraction principles as α-conversion is not observed and the class of accepted programs is not closed under β-reduction. Of these two, the latter issue appears less critical and is shared with other functional intermediate languages that require function arguments to be variables [3, 28]. The violation of α-equivalence is more severe as it precludes equational reasoning. In the second part of this paper we therefore introduce a family of type systems that capture different aspects of the conversion. The most restrictive of these type systems can be used to emit code that satisfies the GNF conditions while lifting the strong syntactic conditions. Given Grail's bijection between imperative registers and functional variables it is not surprising that this type system characterises programs with proper register allocation. It can thus be used as a target for arbitrary register allocation algorithms, and we show that the syntactic GNF conversion can indeed be lifted to a translation between type systems. In Grail, the allocation of registers to program variables amounts to a syntactic transformation on the functional representation. This corresponds to the structure of most imperative compilers which perform register allocation at a low level, after Φ-instructions have been eliminated [21]. In contrast, our framework allows one to study the interactions between optimisations at SSA and machine level, the insertion of compensation code during the Φ-elimination, and low-level register allocation in combination.

Summarising the contributions of this paper, we

$$
\begin{array}{rcl}
b & ::= & e \mid b; f : e \mid b; f : \{b\} \\
e & ::= & \texttt{ret}\ t; \mid \texttt{goto}\ f; \mid \\
 & & x \leftarrow t; e \mid \\
 & & x \leftarrow \Phi(p_1, \ldots, p_n); e \mid \\
 & & \texttt{if}\ t\ \texttt{then}\ e\ \texttt{else}\ e \\
p & ::= & f : t \mid \texttt{start} : t
\end{array}
\qquad
\begin{array}{rcl}
a \in ANF & ::= & t \mid f(t_1, \ldots, t_n) \mid \texttt{let}\ x = t\ \texttt{in}\ a \mid \\
 & & \texttt{if}\ t\ \texttt{then}\ a_1\ \texttt{else}\ a_2 \mid \\
 & & \texttt{rec}\quad f_1\, x_1^1\, \ldots\, x_{n_1}^1 = a_1 \\
 & & \qquad\quad \vdots \\
 & & \qquad\quad f_n\, x_1^n\, \ldots\, x_{n_n}^n = a_n \\
 & & \texttt{in}\ a
\end{array}
$$

$$t \in Term ::= c \mid x$$

Figure 2: Syntax of SSA and ANF

- present a syntactic translation from ANF into GNF whose correctness is stated in terms of a functional operational semantics (Section 2),
- show that the translation corresponds to the elimination of Φ-functions from SSA programs, using a well-known example from the literature as our guiding example
- show that the translation respects the concurrent interpretation of Φ-instructions, using standard examples from the literature (Section 3),
- present a family of type systems where variables that may imperatively be mapped to the same register may inhabit the same type, and introduce a formal code extraction function whose correctness is stated as a preservation result of operational behaviour (Section 4).

We conclude in Section 5 with a discussion of future and related work. This report is an extended version of the workshop paper [6], and contains the proofs of all theorems (Appendix A), as well as some additional material (Sections 3.1, 3.2, 4.3, and 4.4).

2 Syntactic conversion into GNF

2.1 Languages ANF and SSA

Our representations of SSA and ANF are similar to those of [9], but we restrict our attention to a single procedure: function applications in ANF occur as tail calls. Given mutually disjoint sets *Const* of constants (including the special constants **tt** and **ff** and ranged over by $c, d \ldots$) and *Var* of variables (ranged over by $f, g, \ldots, x, y \ldots$), the syntax of SSA and ANF is given in Figure 2. ANF-expressions can be terms, function calls (arguments must be terms), let-bindings of terms, conditionals, and definitions of (possibly mutually recursive) named functions. As is standard practice, we will always assume that function names $f_1, \ldots, f_n$ occurring jointly in a declaration are distinct, and that in each declaration, the formal parameters are distinct and different from the f_i. Furthermore, we only consider first-order programs. Similar assumptions apply to the SSA code: the labelling of jointly defined basic blocks is unique, and the Φ-instructions in a block f carry exactly one argument $g : t$ for each control flow predecessor g of f. The requirement that the formal arguments in each ANF function declaration be distinct means in SSA that all Φ-instructions in a basic block have distinct left-hand sides. This condition is a common requirement in functional languages, and necessary for the concurrent interpretation of all Φ-instructions in a basic block. We refer the reader to [9] for the formal definition of a translation from SSA programs into ANF expressions and a correctness argument for programs which are properly nested, i.e. programs in which the presence of Φ-functions obeys the dominance relation (see also [3] and [15]).

Our operational semantics for ANF is given by a big-step evaluation relation $\mathcal{E} \vdash a \Downarrow v$ where $\mathcal{E}$ is an environment, i.e. a finite map from variables to values. Values are either constants or closures (represented

$$\text{CONST}\frac{}{\mathcal{E} \vdash c \Downarrow c} \qquad \text{VAR}\frac{x \in dom\ \mathcal{E}}{\mathcal{E} \vdash x \Downarrow \mathcal{E}(x)} \qquad \text{LET}\frac{\mathcal{E} \vdash t \Downarrow w \quad \mathcal{E}[x \mapsto w] \vdash a \Downarrow v}{\mathcal{E} \vdash \texttt{let}\ x = t\ \texttt{in}\ a \Downarrow v}$$

$$\text{CALL}\frac{\mathcal{E} \vdash f \Downarrow \langle [x_1,\ldots,x_n], \mathcal{F}, a\rangle \quad \forall i.\ \mathcal{E} \vdash t_i \Downarrow v_i \quad \mathcal{F}[x_i \mapsto v_i]_{i=1,\ldots,n} \vdash a \Downarrow v}{\mathcal{E} \vdash f(t_1,\ldots,t_n) \Downarrow v}$$

$$\text{TRUE}\frac{\mathcal{E} \vdash t \Downarrow \mathbf{tt} \quad \mathcal{E} \vdash a_1 \Downarrow v}{\mathcal{E} \vdash \texttt{if}\ t\ \texttt{then}\ a_1\ \texttt{else}\ a_2 \Downarrow v} \qquad \text{FALSE}\frac{\mathcal{E} \vdash t \Downarrow \mathbf{ff} \quad \mathcal{E} \vdash a_2 \Downarrow v}{\mathcal{E} \vdash \texttt{if}\ t\ \texttt{then}\ a_1\ \texttt{else}\ a_2 \Downarrow v}$$

$$\text{REC}\frac{\mathcal{F} \vdash a \Downarrow v \quad \mathcal{F} = \mathcal{E}[f_i \mapsto \langle [x_1^i,\ldots,x_{n_i}^i], \mathcal{F}, a_i\rangle]_{i=1,\ldots,n}}{\mathcal{E} \vdash \begin{array}{l}\texttt{rec}\quad f_1\ x_1^1\ \ldots\ x_{n_1}^1 = a_1 \\ \qquad\qquad\vdots \\ \qquad\quad f_n\ x_1^n\ \ldots\ x_{n_n}^n = a_n \\ \texttt{in}\ a\end{array} \Downarrow v}$$

Figure 3: Operational semantics of ANF

as triples of formal parameters, environment, and function body):

$$\begin{aligned} \mathcal{C} = \langle [x_1,\ldots,x_n], \mathcal{E}, a\rangle \in Clos &= Var\ list \times Env \times ANF \\ v \in Val &= Const + Clos \\ \mathcal{E} \in Env &= Var \mapsto_{fin} Val \end{aligned}$$

As was pointed out by Milner and Tofte [19], Aczel's theory of non-well-founded sets can be used to justify this setup as it guarantees the existence of the three (mutually recursively defined) semantic categories, and in particular the existence of objects satisfying infinite identities like $\mathcal{F} = \mathcal{E}[x \mapsto \langle [x_1,\ldots,x_n], \mathcal{F}, a\rangle]$. The domain of $\mathcal{E}$ is denoted by $dom\ \mathcal{E}$, and $\mathcal{E}[x \mapsto v]$ represents the environment mapping x to v and acting like $\mathcal{E}$ elsewhere. The rules defining $\mathcal{E} \vdash a \Downarrow v$ are given in Figure 3.
We omit a formal definition of a semantics for SSA programs but recall the standard interpretation of Φ-functions: when the control flow passes from block f to block g, all Φ-instructions

$$\begin{aligned} x_1 &\leftarrow \Phi(f_1^1 : t_1^1, \ldots, f_n^1 : t_n^1) \\ &\vdots \\ x_k &\leftarrow \Phi(f_1^k : t_1^k, \ldots, f_n^k : t_n^k) \end{aligned}$$

in g are interpreted as the concurrent assignment $x_k \leftarrow t_i^k$ where i is the unique index with $f_i = f$.

2.2 Grail normal form and GNF-conversion

For the purpose of this paper, an ANF program is said to be in *Grail normal form* (GNF) if it satisfies the following conditions, where (3) is optional.

1. all functions are fully λ-lifted.
2. for all functions f, all actual arguments in calls to f are variables and coincide syntactically (at each argument position) with the formal parameters in the definition of f.
3. both arms a_1 and a_2 of conditionals $\texttt{if}\ t\ \texttt{then}\ a_1\ \texttt{else}\ a_2$ are either of the form t or $f(t_1,\ldots,t_n)$.

The second condition is also referred to as Grail's "calling convention". In [7] we showed that for programs satisfying the calling convention a functional semantics closely related to $\mathcal{E} \vdash a \Downarrow v$ coincides with a standard imperative semantics. The latter agrees with SSA semantics in the absence of Φ-instructions.

$$\text{G-I}\ \frac{}{(f(t_1,\ldots,t_n),[x_1,\ldots,x_n],x) \rhd f(x_1,\ldots,x_n)}\ \forall i.x_i = t_i$$

$$\text{G-II}\ \frac{(f(t_1,\ldots,t_{i-1},x_i,t_{i+1},\ldots,t_n),[x_1,\ldots,x_n],x) \rhd a}{(f(t_1,\ldots,t_n),[x_1,\ldots,x_n],x) \rhd \texttt{let}\ x_i = t_i\ \texttt{in}\ a}\ \forall j.\ x_i \neq t_j$$

$$\text{G-III}\ \frac{(f(t_1,\ldots,t_n)[x/x_i],[x_1,\ldots,x_n],x) \rhd a}{(f(t_1,\ldots,t_n),[x_1,\ldots,x_n],x) \rhd \texttt{let}\ x = x_i\ \texttt{in}\ a}\ \begin{cases} x_i \neq t_i \\ \forall k.\exists j.\ x_k = t_j \end{cases}$$

Figure 4: Rules defining $\mathcal{G}$

The third condition amounts to requiring ANF expressions to be basic blocks rather than extended basic blocks [21].
A translation of an ordinary ANF program into GNF can be achieved using the following four steps: we

1. α-convert variables globally so that no variable is bound more than once – binding occurs in `let`-statements and function declarations.
2. λ-lift all functions, i.e. turn free variables of function bodies into formal parameters, update the function calls accordingly, and then move all function declarations to the top level. The resulting program contains at most one `rec` statement, at the outermost position.
3. convert each call into code satisfying the calling convention, i.e. ensure that all calls to a function declared by $f\,x_1 \ldots x_n = a$ are *literally* $f(x_1,\ldots,x_n)$.
4. (optionally) normalise branches by inserting fresh function declarations.

The first two steps are well-known, with the two tasks in step (2) often being referred to as parameter lifting and block floating (see for example the work by Danvy et al. [11]). Since our language is first-order, we consider λ-lifting to mean λ-lifting of arguments of ground type throughout the paper.
The purpose of the third step is to implement the effect of a (hypothetical) parallel assignment

$$\texttt{let}\ (x_1,\ldots,x_n) = (t_1,\ldots,t_n)\ \texttt{in}\ f(x_1,\ldots,x_n)$$

by a sequence of unary `let`-bindings followed by the same call $f(x_1,\ldots,x_n)$, where $x_1,\ldots,x_n$ are the formal parameters of f. In principle, this could be achieved by emitting

$$\begin{array}{l} \texttt{let}\,y_1 = t_1\ \texttt{in}\ldots\texttt{let}\,y_n = t_n\ \texttt{in} \\ \texttt{let}\,x_1 = y_1\ \texttt{in}\ldots\texttt{let}\,x_n = y_n\ \texttt{in}\ f(x_1,\ldots,x_n), \end{array}$$

where the temporary variables $y_1,\ldots,y_n$ are distinct and fresh. Instead, we propose the following algorithm $\mathcal{G}$ that uses only a single temporary variable. We define the result of converting $\mathcal{G}(f(t_1,\ldots,t_n),x)$ to be a if

$$(f(t_1,\ldots,t_n),[x_1,\ldots,x_n],x) \rhd a$$

can be derived using the rules given in Figure 4. Again, the x_i are the formal parameters of f's declaration while x is fresh. The result a contains $k+m$ `let`-bindings where k is the number of positions i with $t_i \neq x_i$ and m is the number of cycles (i.e. sequences $[x_{i_0},\ldots,x_{i_{l-1}}]$ of distinct variables from $\{x_1,\ldots,x_n\}$ such that $x_{i_j} = t_{i_{j+1 \bmod l}}$ for $0 \leq j < l$).
The correctness of step (3) may be stated as follows.

Theorem 2.1 *Let* $\mathcal{E}(f) = \langle [x_1,\ldots,x_n], \widehat{\mathcal{E}}, \widehat{a} \rangle$ *and*

$$(f(t_1,\ldots,t_n),[x_1,\ldots,x_n],x) \rhd a.$$

Then for all v, we have $\mathcal{E} \vdash f(t_1,\ldots,t_n) \Downarrow v \iff \mathcal{E} \vdash a \Downarrow v$.

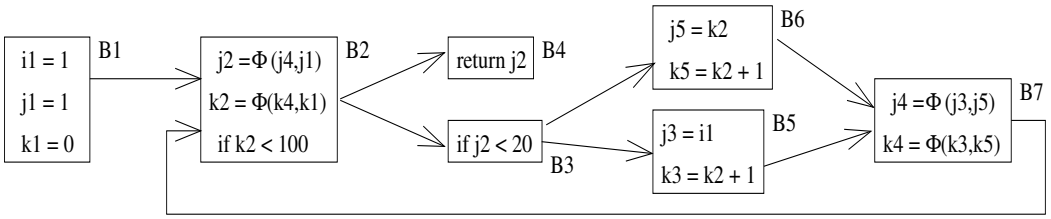


Figure 5: Illustrating Φ-elimination (I) – program taken from [4]

Proof See Appendix A.1.

It is not difficult to see that the side conditions of the rules G-I to G-III are mutually exclusive and exhaustive. All cycles are thus resolved using the same variable x.
Note that the program that results from applying $\mathcal{G}$ to all function calls may violate the property established by the first step, as it may contain variables that are bound at several places.
Finally, step (4) recovers the correspondence between function names and basic block labels. We first replace each non-normalised arm a of a conditional by

$$\texttt{rec}\ f(x_1,\ldots,x_n) = a\ \texttt{in}\ f(x_1,\ldots,x_n)$$

where $x_1,\ldots,x_n$ are the free (ground) variables of a and f is a fresh function identifier. We then repeat λ-lifting. As calls to functions introduced by branch normalisation satisfy the calling convention, programs resulting from converting an ANF program are in GNF, although variables are in general bound at more than one place. In particular, GNF does not respect α-equivalence: renaming a let-bound variable or a formal parameter of a function declaration leads (in general) to code violating the calling condition.

3 GNF conversion is Φ-elimination

We now demonstrate that the GNF conversion corresponds precisely to the elimination of Φ-instructions from SSA code by considering the effect of the four conversion steps on code in SSA form. As running example we use Appel's program [4] (see Figure 5). The ANF representation of this program is

$$\begin{array}{l}
\texttt{let}\ i_1 = 1,\ j_1 = 1, k_1 = 0\ \texttt{in}\\
\texttt{rec}\ f_2\ j_2\ k_2 = \ \texttt{if}\ k_2 < 100\\
\qquad\qquad\qquad \texttt{then}\ \texttt{rec}\ f_7\ j_4\ k_4 = f_2(j_4,k_4)\ \texttt{in}\\
\qquad\qquad\qquad\qquad \texttt{if}\ j_2 < 20\\
\qquad\qquad\qquad\qquad \texttt{then let}\ j_3 = i_1,\ k_3 = k_2 + 1\ \texttt{in}\ f_7(j_3,k_3)\\
\qquad\qquad\qquad\qquad \texttt{else let}\ j_5 = k_2,\ k_5 = k_2 + 1\ \texttt{in}\ f_7(j_5,k_5)\\
\qquad\qquad\qquad \texttt{else}\ j_2\\
\texttt{in}\ f_2(j_1,k_1).
\end{array}$$

Step 1 : Uniqueness of variables holds by the definition of SSA.

Step 2 : Parameter-lifting i_1 in f_2 and f_7 (all other variables are λ-lifted) and performing block-floating yields

$$\begin{array}{l}
\texttt{rec}\ \ f_2\ i_1\ j_2\ k_2 = \ \texttt{if}\ k_2 < 100\\
\qquad\qquad\qquad\qquad \texttt{then}\ \texttt{if}\ j_2 < 20\\
\qquad\qquad\qquad\qquad\qquad \texttt{then let}\ j_3 = i_1,\ k_3 = k_2 + 1\ \texttt{in}\ f_7(j_3,k_3,i_1)\\
\qquad\qquad\qquad\qquad\qquad \texttt{else let}\ j_5 = k_2,\ k_5 = k_2 + 1\ \texttt{in}\ f_7(j_5,k_5,i_1)\\
\qquad\qquad\qquad\qquad \texttt{else}\ j_2\\
\qquad f_7\ j_4\ k_4\ i_1 = f_2(i_1,j_4,k_4)\\
\texttt{in let}\ i_1 = 1,\ j_1 = 1, k_1 = 0\ \texttt{in}\ f_2(i_1,j_1,k_1).
\end{array}$$

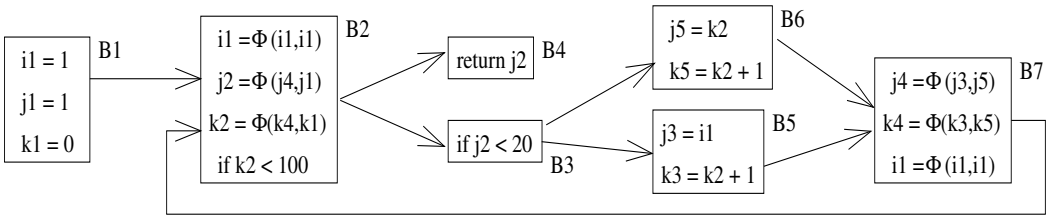


Figure 6: Illustrating Φ-elimination (II): λ-lifting

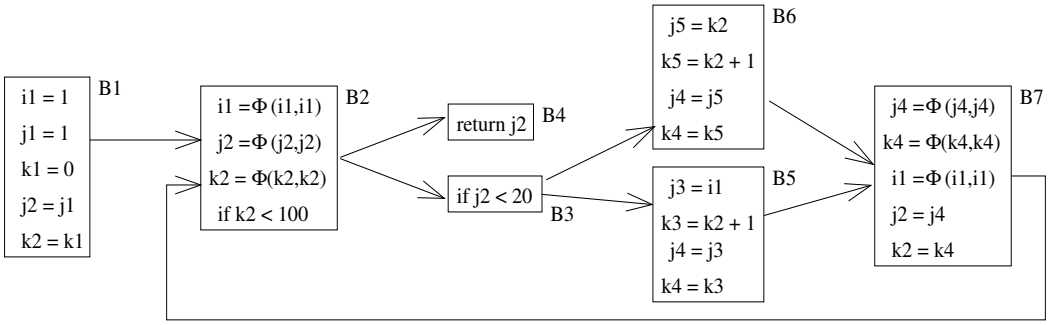


Figure 7: Illustrating Φ-elimination (III): conversion $\mathcal{G}$

In SSA, λ-lifting amounts to the insertion of trivial Φ-instructions $x = \Phi(x,\ldots,x)$ - see Figure 6, where blocks 2 and 7 contain the trivial Φ-instruction $i_1 = \Phi(i_1,i_1)$. Note that the result is not in SSA form any longer as variable i_1 now has three sites of definition.

Step 3 : The fact that λ-lifted variables automatically satisfy the calling restriction is respected by $\mathcal{G}$ as no trivial `let`-bindings `let` $i_1 = i_1$ `in`... are introduced.

$$
\begin{array}{l}
\texttt{rec } f_2\ i_1\ j_2\ k_2 = \\
\qquad \texttt{if } k_2 < 100 \\
\qquad \texttt{then if } j_2 < 20 \\
\qquad\qquad \texttt{then let } j_3 = i_1,\ k_3 = k_2 + 1,\ j_4 = j_3, k_4 = k_3 \texttt{ in } f_7(j_4,k_4,i_1) \\
\qquad\qquad \texttt{else let } j_5 = k_2,\ k_5 = k_2 + 1,\ j_4 = j_5, k_4 = k_5 \texttt{ in } f_7(j_4,k_4,i_1) \\
\qquad \texttt{else } j_2 \\
\quad f_7\ j_4\ k_4\ i_1 = \texttt{let } j_2 = j_4, k_2 = k_4 \texttt{ in } f_2(i_1,j_2,k_2) \\
\texttt{in let } i_1 = 1,\ j_1 = 1, k_1 = 0,\ j_2 = j_1, k_2 = k_1 \texttt{ in } f_2(i_1,j_2,k_2)
\end{array}
$$

In SSA, the effect of $\mathcal{G}$ is to make all Φ-functions trivial – see Figure 7.

Step 4 : Branch-normalisation first introduces parameter-lifted definitions for functions f_3, f_5 and f_6. Since these functions have only one call site it is always possible to name the parameters such that the calling restriction is respected.

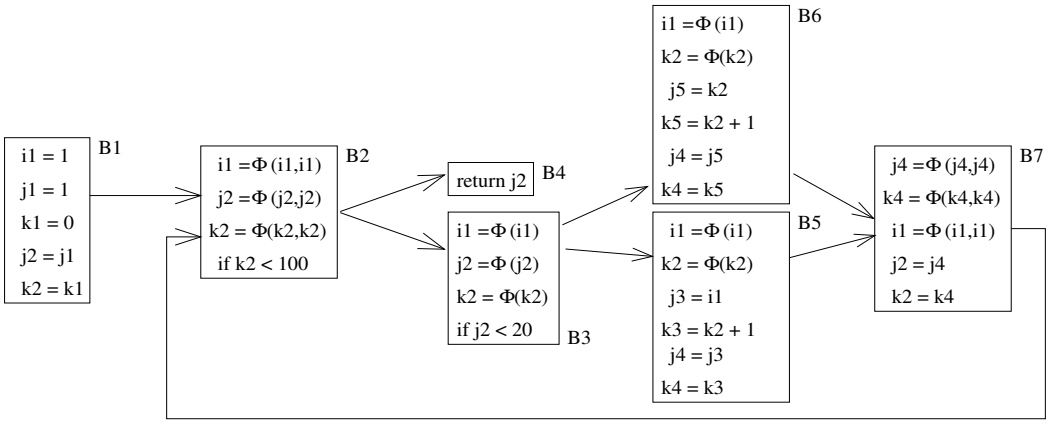


Figure 8: Illustrating Φ-elimination (IV a): branch normalisation (part 1)

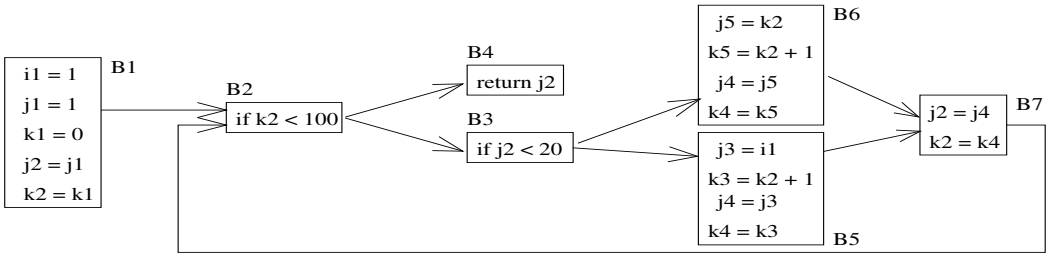


Figure 9: Illustrating Φ-elimination (IV b): branch normalisation (part 2)

$$
\begin{array}{l}
\texttt{rec}\ \ f_2\ i_1\ j_2\ k_2 = \ \texttt{if}\ k_2 < 100 \\
\qquad\qquad\qquad\quad \texttt{then}\ \ \texttt{rec}\ f_3\ i_1\ j_2\ k_2 = \\
\qquad\qquad\qquad\qquad\quad \texttt{if}\ j_2 < 20 \\
\qquad\qquad\qquad\qquad\quad \texttt{then}\ \ \texttt{rec}\ f_5\ i_1\ k_2 = \ \texttt{let}\ \ j_3 = i_1,\ k_3 = k_2 + 1, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad j_4 = j_3,\ k_4 = k_3\ \texttt{in}\ f_7(j_4,k_4,i_1) \\
\qquad\qquad\qquad\qquad\qquad\quad \texttt{in}\ f_5(i_1,k_2) \\
\qquad\qquad\qquad\qquad\quad \texttt{else}\ \ \texttt{rec}\ f_6\ i_1\ k_2 = \ \texttt{let}\ \ j_5 = k_2,\ k_5 = k_2 + 1, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad j_4 = j_5, k_4 = k_5\ \texttt{in}\ f_7(j_4,k_4,i_1) \\
\qquad\qquad\qquad\qquad\qquad\quad \texttt{in}\ f_6(i_1,k_2) \\
\qquad\qquad\qquad\qquad\ \ \texttt{in}\ f_3(i_1\ j_2\ k_2) \\
\qquad\qquad\qquad\quad \texttt{else}\ j_2 \\
\qquad f_7\ j_4\ k_4\ i_1 = \texttt{let}\ j_2 = j_4, k_2 = k_4\ \texttt{in}\ f_2(i_1,j_2,k_2) \\
\texttt{in}\ \texttt{let}\ i_1 = 1,\ j_1 = 1, k_1 = 0,\ j_2 = j_1, k_2 = k_1\ \texttt{in}\ f_2(i_1,j_2,k_2)
\end{array}
$$

In SSA, this amounts to inserting unary trivial Φ-instructions in $B3$, $B5$ and $B6$ – see Figure 8. The final phase, λ-lifting, moves functions f_3, f_5 and f_6 to the top level and results in

$$
\begin{array}{l}
\texttt{rec}\ \ f_2\ i_1\ j_2\ k_2 = \texttt{if}\ k_2 < 100\ \texttt{then}\ f_3(i_1,j_2,k_2)\ \texttt{else}\ j_2 \\
\qquad f_3\ i_1\ j_2\ k_2 = \texttt{if}\ j_2 < 20\ \texttt{then}\ f_5(i_1,k_2)\ \texttt{else}\ f_6(i_1,k_2) \\
\qquad f_5\ i_1\ k_2 = \ \texttt{let}\ j_3 = i_1,\ k_3 = k_2 + 1,\ j_4 = j_3, k_4 = k_3\ \texttt{in}\ f_7(j_4,k_4,i_1) \\
\qquad f_6\ i_1\ k_2 = \ \texttt{let}\ j_5 = k_2,\ k_5 = k_2 + 1,\ j_4 = j_5, k_4 = k_5\ \texttt{in}\ f_7(j_4,k_4,i_1) \\
\qquad f_7\ j_4\ k_4\ i_1 = \texttt{let}\ j_2 = j_4, k_2 = k_4\ \texttt{in}\ f_2(i_1,j_2,k_2) \\
\texttt{in}\ \texttt{let}\ i_1 = 1,\ j_1 = 1, k_1 = 0,\ j_2 = j_1, k_2 = k_1\ \texttt{in}\ f_2(i_1,j_2,k_2)
\end{array}
$$

Figure 9 shows the result of deleting the (trivial) Φ-functions.

GNF conversion models Φ-elimination by inserting compensation code immediately prior to jumps. Although no trival assigments $\texttt{let}\ x = x\ \texttt{in}\ \ldots$ are inserted, the resulting code could be further optimised

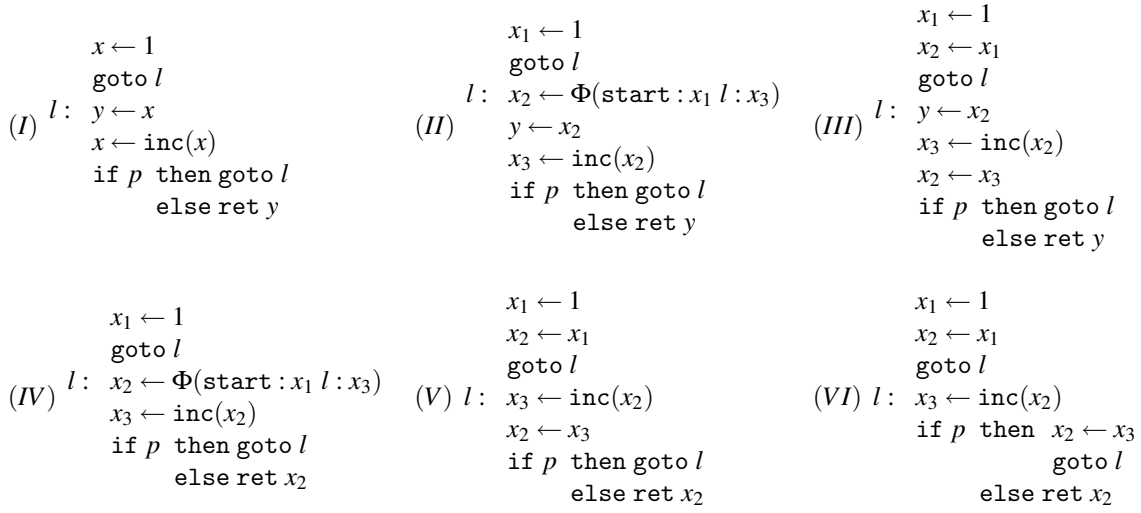


Figure 10: The *lost copy* problem (taken from [8])

by coalescing inserted code with instructions that are already present in the same basic block. Furthermore, GNF conversion does not require programs to be in *edge-split* form[1]. Like edge-splitting, the normalisation of a branch only adds a single (jump) instruction, but in contrast to edge-splitting, it is performed as the optional last step of our transformation.
We now demonstrate that our algorithm respects the concurrent interpretation of Φ-instructions, using standard examples from the literature [8].

3.1 Lost copies and swaps

Two issues in the literature on the elimination of Φ-instructions are the *lost copy problem* and the *swap problem* [8]. Both problems are treated correctly by our algorithm as may be illustrated by considering the intermediate program optimisation technique *constant folding*.
For the lost copy problem consider the code shown in column (I) in Figure 10. Converting this code into SSA form yields the code shown in column (II), and eliminating the Φ-instruction by inserting compensation code in both predecessor basic blocks according to [10] results in column (III). The code coincides semantically with the original program, despite the fact that the code in column (II) was not in edge-split form. In contrast, applying constant folding to (II) yields the code shown in column (IV), and eliminating the Φ-instruction now leads to code that is semantically different from the original program (column (V)): on leaving the loop the value of x_2 is that of the *last* iteration rather than that of the *penultimate* iteration. Finally, column (VI) shows that edge-splitting the copy-folded code in column (IV) before eliminating the Φ-instruction results in correct code. The transformation $\mathcal{G}$ inserts any compensating instructions in the branches of conditionals and thus yields the correct result when applied to the code of column (IV).
The swap problem occurs if a variable to which the result of a Φ-instruction is assigned also occurs as an argument of another Φ-instruction in the same basic block, as is illustrated in Figure 11. Converting the code shown in column (I) into SSA and folding copies leads to the code shown in column (II). Eliminating the Φ-instructions by inserting two appropriate compensating instructions in both preceding basic blocks leads to semantically different code as the variables get identified (column (III)). The code remains incorrect if (II) is converted into edge-split form or the Φ-instructions are eliminated in a

[1] A program is in edge-split form if no control flow edge links a block with out-degree greater than one to a block with in-degree greater than one.

```
    (I)              (II)                (III)              (IV)
    a ← ...          a1 ← ...            a1 ← ...           a1 ← ...
    b ← ...          b1 ← ...            b1 ← ...           b1 ← ...
    goto l           goto l              a2 ← a1            a2 ← a1
l:  x ← a        l:  a2 ← Φ(start:a1 l:b2)  b2 ← b1         b2 ← b1
    a ← b            b2 ← Φ(start:b1 l:a2)  goto l          goto l
    b ← x            if p                l:  a2 ← b2    l:  if p
    if p             then goto l             b2 ← a2        then x ← a2
    then goto l      else ...                if p                a2 ← b2
    else ...                                 then goto l         b2 ← x
                                             else ...            goto l
                                                            else ...
```

Figure 11: The *swap* problem (taken from [8])

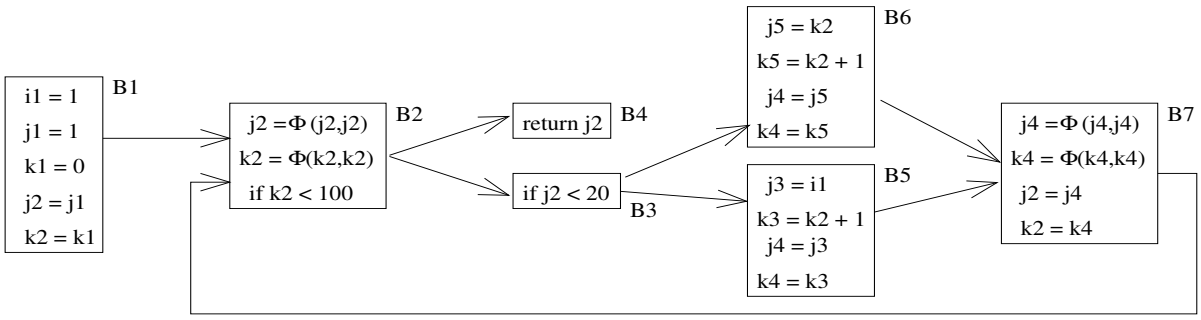


Figure 12: Illustrating Φ-elimination (IV): performing $\mathcal{G}$ prior to λ-lifting

different order. Again, the conversion $\mathcal{G}$ detects the mutual dependence between the two Φ-instructions and emits the correct code, shown in column (IV).

3.2 Alternative conversion routes

While the relative order of α-conversion, parameter-lifting and block-floating is fixed, the conversion $\mathcal{G}$ may also by performed prior to λ-lifting. For example, applying $\mathcal{G}$ to Appel's original code yields

```
let i1 = 1, j1 = 1, k1 = 0 in
rec f2 j2 k2 =  if k2 < 100
                then rec f7 j4 k4 = let j2 = j4, k2 = k4 in f2(j2,k2) in
                     if j2 < 20
                     then let j3 = i1, k3 = k2 + 1, j4 = j3, k4 = k3 in f7(j4,k4)
                     else let j5 = k2, k5 = k2 + 1, j4 = j5, k4 = k5 in f7(j4,k4)
                else j2
in let j2 = j1, k2 = k1 in f2(j2,k2)
```

The corresponding SSA code (Figure 12) contains trivial Φ-instructions for the subscripted j's and k's, but not for i_1. If we now apply λ-lifting, the additional trivial Φ-instructions for i_1 are inserted – in fact the result is exactly that of Figure 7. Phases (2) and (3) may be swapped in general since any variable which needs to be λ-lifted still has only one site of definition, even after $\mathcal{G}$ has been applied (it does not occur as a function parameter).

Since $\mathcal{G}$ may be applied to programs which contain variables with more than one site of declaration, one may be tempted to postpone α-conversion until $\mathcal{G}$ has been applied. However, α-conversion in general destroys the calling convention, hence performing $\mathcal{G}$ as the first transformation step would yield incorrect results.

$$\text{Const}\frac{}{\Gamma \vdash_\Sigma c : \rho} \qquad \text{Let}\frac{\Gamma \vdash_\Sigma t : \rho \quad \Gamma, x : \rho_1 \vdash_\Sigma a : \rho_2 \quad x \notin dom\,\Sigma}{\Gamma \vdash_\Sigma \texttt{let}\ x = t\ \texttt{in}\ a : \rho_2}$$

$$\text{Var}\frac{\Gamma!x = \rho}{\Gamma \vdash_\Sigma x : \rho} \qquad \text{If}\frac{\Gamma \vdash_\Sigma t : \rho \quad \Gamma \vdash_\Sigma a_1 : \rho_1 \quad \Gamma \vdash_\Sigma a_2 : \rho_1}{\Gamma \vdash_\Sigma \texttt{if}\ t\ \texttt{then}\ a_1\ \texttt{else}\ a_2 : \rho_1}$$

$$\text{Call}\frac{\Sigma(f) = (\rho_1, \ldots, \rho_n) \to \rho \quad \forall i \in \{1, \ldots, n\}. \Gamma \vdash_\Sigma t_i : \rho'_i}{\Gamma \vdash_\Sigma f(t_1, \ldots, t_n) : \rho}$$

$$\text{Rec}\frac{\begin{array}{ll} \Gamma \vdash_\Pi \widehat{a} : \rho & \forall i.\ \Gamma, x^i_1 : \rho^i_1, \ldots, x^i_{n_i} : \rho^i_{n_i} \vdash_\Pi a_i : \rho_i \\ \forall i\ j\ k.\ j \neq k \Rightarrow \rho^i_j \neq \rho^i_k & \forall i.\ f_i \notin dom\,\Gamma \cup dom\,\Sigma \\ \forall i\ j.\ x^i_j \notin dom\,\Pi & \Pi = \Sigma[f_i \mapsto (\rho^i_1, \ldots, \rho^i_{n_i}) \to \rho_i]_{i=1,\ldots,n} \end{array}}{\Gamma \vdash_\Sigma \texttt{rec}\ [f_i\ x^i_1 \ldots x^i_{n_i} = a_i]_{i=1,\ldots,n}\ \texttt{in}\ a : \rho}$$

Figure 13: Rules for type system $\mathfrak{T}$

4 Type systems for register allocation

In this section, we introduce a family of type systems for register allocation, including one variant that captures Grail's calling convention. None of the systems requires function arguments to syntactically coincide with the formal parameters. We prove the operational correctness of register allocation, i.e. the fact that replacing each register type occurring in a derivation with a fresh variable does not affect the outcome of evaluating the program. Finally, we relate the variations, and observe that the most restrictive system corresponds to GNF.

4.1 Type system

The type systems use sequents of the form $\Gamma \vdash_\Sigma a : \tau$ where types τ are built from register types ρ (which range over an abstract class *Regs* of register identifiers) using the grammar

$$\tau \in \mathit{Type} ::= \rho \mid (\rho_1, \ldots, \rho_n) \to \rho.$$

Register contexts Γ are *lists* $x_1 : \rho_1, \ldots, x_n : \rho_n$ where the order of entries arises from the order of assignments, and signatures Σ are partial maps from variables to first-order types $(\rho_1, \ldots, \rho_n) \to \rho$. For register contexts we define the non-standard lookup operation $\Gamma!x$ by

$$[\,]!x = \mathit{undefined} \text{ and } \Gamma, y : \rho\,!x = \begin{cases} \rho & \text{if } x = y \\ \Gamma!x & \text{if } x \neq y \text{ and } \Gamma!x \neq \rho \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

while $dom\,\Gamma = \{x | \exists \rho.\ \Gamma!x = \rho\}$, $cod\,\Gamma = \{\rho | \exists x.\ \Gamma!x = \rho\}$ and all operations on signatures are defined as usual. Notice that in particular, $\Gamma, x : \rho, \Delta!x = \rho$ implies $\rho \notin cod\,\Delta$. The typing rules for the first type system, $\mathfrak{T}$, are given in Figure 13. Two rules require some explanations. In rule Var, the side condition $\Gamma!x = \rho$ retrieves register variables from the context in a last-in-first-out fashion. This guarantees that whenever two variables are mapped to the same register, only the most recently written variable is accessible. In rule Let, the result of evaluating the term t may may be stored in an arbitrary register. We do not require ρ_1 and ρ to be identical – indeed, the case where t is a variable and $\rho_1 \neq \rho$ holds corresponds to a register move. Also notice that $\Gamma, x : \rho_1$ arises by extending Γ at its right-most position, again enforcing the LIFO behaviour of contexts.
As an example, Figure 14 shows the derivation of

$$\Gamma \vdash_\Sigma \texttt{let}\ x = 5\ \texttt{in}\ \texttt{let}\ y = x\ \texttt{in}\ y : \rho$$

$$\dfrac{\dfrac{}{\Gamma \vdash_\Sigma 5 : \rho} \qquad \dfrac{\dfrac{\Gamma, x:\rho \,!\, x = \rho}{\Gamma, x:\rho \vdash_\Sigma x : \rho} \qquad \dfrac{\Gamma, x:\rho, y:\rho \,!\, y = \rho}{\Gamma, x:\rho, y:\rho \vdash_\Sigma y : \rho}}{\Gamma, x:\rho \vdash_\Sigma \texttt{let}\ y = x\ \texttt{in}\ y : \rho}}{\Gamma \vdash_\Sigma \texttt{let}\ x = 5\ \texttt{in}\ \texttt{let}\ y = x\ \texttt{in}\ y : \rho}$$

Figure 14: Example typing derivation

for arbitrary Γ and $\{x, y\} \cap \Sigma = \emptyset$, demonstrating that registers may be reused as soon as their previous content becomes dead, and that registers may be shared between the source and the target of a move. In both applications of Let, the source of the asssignment and the target are given identical types.

Definition A signature Σ is well-formed if for all $x \in dom\ \Sigma$ with $\Sigma(x) = (\rho_1, \ldots, \rho_n) \to \rho$, the ρ_i are distinct. A derivation $\mathcal{D} : \Gamma \vdash_\Sigma a : \tau$ is well-formed if Σ is well-formed, and $dom\ \Gamma \cap dom\ \Sigma = \emptyset$.

Both conditions mentioned in this definition propagate upwards through all typing rules and thus hold for any sequent of a well-formed derivation.
In addition to the type system $\mathfrak{T}$, we consider several variations. These are obtained by imposing one or both of the additional side conditions

$$\forall i \in \{1, \ldots, n\}.\ t_i = y_i \text{ and} \tag{1}$$
$$\forall i \in \{1, \ldots, n\}.\ \rho_i' = \rho_i \tag{2}$$

on rule Call, and are denoted by $\mathfrak{T}_x$ (condition (1)), $\mathfrak{T}_\rho$ (condition (2)), and $\mathfrak{T}_{x,\rho}$ (both conditions). The four calculi can be ordered by restrictiveness into the diamond $\mathfrak{T} \sqsubset \mathfrak{T}_x$, $\mathfrak{T}_x \sqsubset \mathfrak{T}_{x,\rho}$, $\mathfrak{T} \sqsubset \mathfrak{T}_\rho$, $\mathfrak{T}_\rho \sqsubset \mathfrak{T}_{x,\rho}$ ($\mathfrak{T}_x$ and $\mathfrak{T}_\rho$ are incomparable). Of particular interest are $\mathfrak{T}_\rho$ and $\mathfrak{T}_{x,\rho}$ as condition (2) requires arguments to be available in the registers specified by $\Sigma(x)$. It thus corresponds to Grail's calling convention: a call $x(a,b)$ to a function with $\Sigma(x) = (\mathtt{r1}, \mathtt{r2}) \to \mathtt{r2}$ is well-typed exactly if $\Gamma \vdash_\Sigma a : \mathtt{r1}$ and $\Gamma \vdash_\Sigma b : \mathtt{r2}$ hold, where Definition 4.1 ensures $\mathtt{r1} \neq \mathtt{r2}$.

Example Consider two function definitions for the factorial function.

```
fac1 n a =  let test = n < 1 in
            if test then a else let m = n − 1, b = a ∗ n in fac1(m,b)
fac2 n a =  let test = n < 1 in
            if test then a else let b = a ∗ n, m = n − 1 in fac2(m,b)
```

For `fac1` no well-formed typing exists in system $\mathfrak{T}_\rho$, since $\Sigma(\mathtt{fac1}) = (\rho_\mathtt{n}, \rho_\mathtt{a}) \to \rho$ yields $\rho_\mathtt{n} \neq \rho_\mathtt{a}$, so from rule Call we obtain $\rho_\mathtt{n} = \rho_\mathtt{m}$ and $\rho_\mathtt{a} = \rho_\mathtt{b}$, while the typing of the `else`-branch requires us to derive $\mathtt{n} : \rho_\mathtt{n}, \mathtt{a} : \rho_\mathtt{a}, \mathtt{test} : \rho_\mathtt{test}, \mathtt{m} : \rho_\mathtt{m} \vdash \mathtt{n} : \rho$ for some ρ, hence $\rho_\mathtt{n} \neq \rho_\mathtt{m}$. In contrast, program `fac2` is typeable using the three registers $\rho_\mathtt{n} = \rho_\mathtt{m}$, $\rho_\mathtt{a} = \rho_\mathtt{b}$ and $\rho_\mathtt{test}$.

Unless stated otherwise, statements in the remainder of this paper refer to the system $\mathfrak{T}$ and are thus independent of the additional side conditions.

4.2 Register allocation

Each well-formed typing derivation for a program a uniquely determines a register-allocated program which is obtained by choosing a fresh variable for each register identifier ρ that occurs in the derivation and converting a so that any binding of a variable x of type ρ is replaced by the fresh variable associated to ρ.

$$\mathcal{A}_\alpha(\ \text{Const}\ \frac{}{\Gamma \vdash_\Sigma c : \rho}\ ) \quad = \quad c$$

$$\mathcal{A}_\alpha(\ \text{Var}\ \frac{\Gamma!x = \rho}{\Gamma \vdash_\Sigma x : \rho}\ ) \quad = \quad \alpha(\rho)$$

$$\mathcal{A}_\alpha(\ \text{Let}\ \frac{\begin{array}{c}\mathcal{D}_1 : \Gamma \vdash_\Sigma t : \rho \\ \mathcal{D}_2 : \Gamma, x : \rho_1 \vdash_\Sigma a : \rho_2 \\ x \notin dom\,\Sigma\end{array}}{\Gamma \vdash_\Sigma \texttt{let}\ x = t\ \texttt{in}\ a : \rho_2}\ ) \quad = \quad \texttt{let}\ \alpha(\rho_1) = \mathcal{A}_\alpha(\mathcal{D}_1)\ \texttt{in}\ \mathcal{A}_\alpha(\mathcal{D}_2)$$

$$\mathcal{A}_\alpha(\text{If}\ \frac{\mathcal{D}_0 : \Gamma \vdash_\Sigma t : \rho \quad \forall i \in \{1,2\}.\ \mathcal{D}_i : \Gamma \vdash_\Sigma a_i : \rho_1}{\Gamma \vdash_\Sigma \texttt{if}\ t\ \texttt{then}\ a_1\ \texttt{else}\ a_2 : \rho_1}\ ) \quad = \quad \left\{\begin{array}{ll}\texttt{if}\ \mathcal{A}_\alpha(\mathcal{D}_0) & \texttt{then}\ \mathcal{A}_\alpha(\mathcal{D}_1) \\ & \texttt{else}\ \mathcal{A}_\alpha(\mathcal{D}_2)\end{array}\right.$$

$$\mathcal{A}_\alpha(\ \text{Call}\ \frac{\Sigma(f) = (\rho_1, \ldots, \rho_n) \to \rho \quad \forall i \in \{1,\ldots,n\}.\mathcal{D}_i : \Gamma \vdash_\Sigma t_i : \rho'_i}{\Gamma \vdash_\Sigma f(t_1, \ldots, t_n) : \rho}\ ) \quad = \quad f(\mathcal{A}_\alpha(\mathcal{D}_1), \ldots, \mathcal{A}_\alpha(\mathcal{D}_n))$$

$$\mathcal{A}_\alpha(\ \text{Rec}\ \frac{\begin{array}{l}\mathcal{D}_0 : \Gamma \vdash_\Pi \widehat{a} : \rho \\ \forall i.\ \mathcal{D}_i : \Gamma, x^i_1 : \rho^i_1, \ldots, x^i_{n_i} : \rho^i_{n_i} \vdash_\Pi a_i : \rho_i \\ \forall i\ j\ k.\ j \neq k \Rightarrow \rho^i_j \neq \rho^i_k \\ \forall i.\ f_i \notin dom\,\Gamma \cup dom\,\Sigma \\ \forall i\ j.\ x^i_j \notin dom\,\Pi \\ \Pi = \Sigma[f_i \mapsto (\rho^i_1, \ldots, \rho^i_{n_i}) \to \rho_i]_{i=1,\ldots,n}\end{array}}{\Gamma \vdash_\Sigma \texttt{rec}\ [f_i\ x^i_1 \ldots x^i_{n_i} = a_i]_i\ \texttt{in}\ a : \rho}\ ) \quad = \quad \left\{\begin{array}{l}\texttt{rec} \\ [f_i\ \alpha(\rho^i_I) \ldots \alpha(\rho^i_{n_i}) = \mathcal{A}_\alpha(\mathcal{D}_i)]_i \\ \texttt{in}\ \mathcal{A}_\alpha(\mathcal{D}_0)\end{array}\right.$$

Figure 15: Register allocation: rewriting step

Definition An injective map $\alpha : Regs \to Var$ is called a *register allocation* for $\mathcal{D}$ if all sequents $\Gamma \vdash_\Sigma a : \tau$ in $\mathcal{D}$ satisfy $cod\,\alpha \cap dom\,\Sigma = \emptyset$.

Thus, an allocation for $\mathcal{D}$ is also an allocation for any subderivation $\mathcal{D}'$ of $\mathcal{D}$.
The rewriting step is defined by the function $\mathcal{A}_\alpha(.)$ in Figure 15, which converts a typing derivation $\mathcal{D}$ with final sequent $\Gamma \vdash_\Sigma a : \rho$ into $\mathcal{A}_\alpha(a)$, given allocation α.

Example For an allocation α with $\alpha(\rho_{\mathtt{n}}) = \alpha(\rho_{\mathtt{m}}) = \mathtt{n}$, $\alpha(\rho_{\mathtt{a}}) = \alpha(\rho_{\mathtt{b}}) = \mathtt{a}$, and $\alpha(\rho_{\mathtt{test}}) = \mathtt{test}$, the result of applying $\mathcal{A}_\alpha(.)$ to `fac2` is

$$\begin{array}{lll}\mathtt{fac2'\ n\ a} = & \mathtt{let\ test} & = \mathtt{n < 1\ in} \\ & \mathtt{if\ test\ then\ a\ else\ let\ a} & = \mathtt{a * n,\ n} = \mathtt{n - 1\ in\ fac2'(n, a).}\end{array}$$

This code behaves exactly like `fac2` but is in (non branch normalised) GNF.

The program $\mathcal{A}_\alpha(\mathcal{D})$ again corresponds to the result of eliminating Φ-instructions and satisfies Grail's calling convention. It also coincides semantically with a, as we now show.

4.3 Operational correctness

We will now show that $\mathcal{A}_\alpha(\mathcal{D})$ is operationally equivalent to a if α is a register allocation for well-typed $\mathcal{D} : \Gamma \vdash_\Sigma a : \tau$. Since register allocation renames variables, we first introduce equivalences for environments that are defined over different domains.
Consider the set $U := Val \times Type \times Val$, the powerset $\mathcal{P}(U)$ and the operator $F : \mathcal{P}(U) \to \mathcal{P}(U)$ defined in Figure 16, where α is a fixed allocation. It is not difficult to see that F is monotone, i.e. that $Q \subseteq P$ implies $F(Q) \subseteq F(P)$. Since $\mathcal{P}(U)$ is a complete lattice, F has unique least and greatest fixed points $Q^{min} := \cap\{Q \mid F(Q) \subseteq Q\}$ and $Q^{max} := \cup\{Q \mid Q \subseteq F(Q)\}$. The latter one is now used to define equivalences on values and environments.

$F(Q) =$
$\{(v,\tau,w) \mid$ if $\tau = \rho$ then $v = w = c$ for some c
if $\tau = (\rho_1,\ldots,\rho_n) \to \rho$ then there are $x_i, \mathcal{E}_i$ and a_i such that $v = \langle [x_1,\ldots,x_n], \mathcal{E}_1, a_1\rangle$ and
$w = \langle [\alpha(\rho_1),\ldots,\alpha(\rho_n)], \mathcal{E}_2, a_2\rangle, i \neq j$ implies $x_i \neq x_j$ and $\alpha(\rho_i) \neq \alpha(\rho_j)$, and there are Γ and Σ
such that $\Delta := \Gamma, x_1 : \rho_1, \ldots, x_n : \rho_n$ and $x \in fv(a_1) \setminus \{x_1,\ldots,x_n\}$ satisfy the following four items:
I) $\mathcal{D} : \Delta \vdash_\Sigma a_1 : \rho$ is a well-formed derivation, α is an allocation for $\mathcal{D}$ and $a_2 = \mathcal{A}_\alpha(\mathcal{D})$
II) $x \in dom\ \mathcal{E}_1 \cap (dom\ \Delta \cup dom\ \Sigma)$
III) if $x \in dom\ \Delta$ then $\alpha(\Delta!x) \in dom\ \mathcal{E}_2 \setminus \{\alpha(\rho_1),\ldots,\alpha(\rho_n)\}$ and $(\mathcal{E}_1(x), \Delta!x, \mathcal{E}_2(\alpha(\Delta!x))) \in Q$
IV) if $x \in dom\ \Sigma$ then $(\mathcal{E}_1(x), \Sigma(x), \mathcal{E}_2(x)) \in Q$.

Figure 16: Definition of operator $F : \mathcal{P}(U) \to \mathcal{P}(U)$.

Definition We define the relations $. \sim_. .$ and $. \approx_.^{(.,.)} .$ by

$$v \sim_\tau w \iff (v,\tau,w) \in Q^{max} \text{ and}$$

$$\mathcal{E} \approx_V^{(\Gamma,\Sigma)} \mathcal{F} \iff \begin{cases} \text{i) } V \subseteq dom\ \mathcal{E} \cap (dom\ \Gamma \cup dom\ \Sigma), \\ \text{ii) } \forall x \in V \cap dom\ \Gamma.\ \mathcal{E}(x) \sim_{\Gamma!x} \mathcal{F}(\alpha(\Gamma!x)), \text{ and} \\ \text{iii) } \forall x \in V \cap dom\ \Sigma.\ \mathcal{E}(x) \sim_{\Sigma(x)} \mathcal{F}(x). \end{cases}$$

Lemma 1 *If $\Gamma \vdash_\Sigma a : \tau$ then $fv(a) \subseteq dom\ \Gamma \cup dom\ \Sigma$.*

The following soundness result is proven by structural induction on the operational semantics.

Theorem 4.1 *Let $\mathcal{D} : \Gamma \vdash_\Sigma a : \tau$ be well-formed, α a register allocation for $\mathcal{D}$ and $\mathcal{E} \approx_{fv(a)}^{(\Gamma,\Sigma)} \mathcal{F}$. Then*

- *$\mathcal{E} \vdash a \Downarrow v$ implies $\mathcal{F} \vdash \mathcal{A}_\alpha(\mathcal{D}) \Downarrow w$ for some w with $v \sim_\tau w$, and*
- *$\mathcal{F} \vdash \mathcal{A}_\alpha(\mathcal{D}) \Downarrow w$ implies $\mathcal{E} \vdash a \Downarrow v$ for some v with $v \sim_\tau w$.*

Proof See Appendix A.2.

In particular, for a top-level program a with $fv(a) = \emptyset$, the equivalence $\mathcal{E} \approx_{fv(a)}^{(\Gamma,\Sigma)} \mathcal{F}$ is easily seen to be fulfilled, so $\mathcal{A}_\alpha(\mathcal{D})$ is operationally equivalent to a.

4.4 Variations of the type system

A type system for register allocated parameter-lifted programs can be obtained by replacing the side condition

$$\forall i.\ \Gamma, x_1^i : \rho_1^i, \ldots, x_{n_i}^i : \rho_{n_i}^i \vdash_\Pi a_i : \rho_i \tag{3}$$

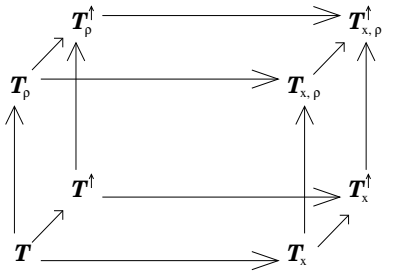
of rule Rec by

$$\forall i.\ x_1^i : \rho_1^i, \ldots, x_{n_i}^i : \rho_{n_i}^i \vdash_\Pi a_i : \rho_i \ \wedge\ fv(a_i) \subseteq \{x_1^i,\ldots,x_{n_i}^i\} \cup dom\ \Pi. \tag{4}$$

Variables accessed within function bodies are required to be amongst the formal parameters. The second part of condition (4) models the fact that we are only interested in parameter-lifting register variables. Again, four variations (denoted by $\mathfrak{T}^\uparrow$, $\mathfrak{T}_x^\uparrow$, $\mathfrak{T}_\rho^\uparrow$, and $\mathfrak{T}_{x,\rho}^\uparrow$) arise by combining parameter lifting with the conditions on the syntactic form or the typing of function arguments. The diamond structure relating the earlier four type systems also exists for parameter lifted programs, and by considering the monotone operator $F : \mathcal{P}(U) \to \mathcal{P}(U)$ given in Figure 17, Theorem 4.1 can be proven for $\mathfrak{T}^\uparrow$ and hence for the all type systems that satisfy condition (4). The new definition of F arises by requiring that the context Γ mentioned in Figure 16 be empty and adding the restriction on free variables from condition (4). Typeability using (4) entails typeability using (3), so we obtain a cube

$F(Q) = \{(v,\tau,w) \mid$ if $\tau = \rho$ then $v = w = c$ for some c
if $\tau = (\rho_1,\ldots,\rho_n) \to \rho$ then there are $x_i, \mathcal{E}_i$ and a_i such that $v = \langle [x_1,\ldots,x_n], \mathcal{E}_1, a_1\rangle$,
$w = \langle [\alpha(\rho_1),\ldots,\alpha(\rho_n)], \mathcal{E}_2, a_2\rangle, i \neq j$ implies $x_i \neq x_j$ and $\alpha(\rho_i) \neq \alpha(\rho_j)$, and there is a Σ
such that $\Delta := x_1 : \rho_1,\ldots,x_n : \rho_n$ satisfies the following three items:
I) $fv(a_1) \subseteq dom\,\Delta \cup dom\,\Sigma$
II) $\mathcal{D} : \Delta \vdash_\Sigma a_1 : \rho$ is a well-formed derivation, α is an allocation for $\mathcal{D}$, and $a_2 = \mathcal{A}_\alpha(\mathcal{D})$,
III) for $x \in fv(a_1) \setminus \{x_1,\ldots,x_n\}$ we have $x \in dom\,\mathcal{E}_1 \cap dom\,\Sigma$ and $(\mathcal{E}_1(x), \Sigma(x), \mathcal{E}_2(x)) \in Q$.

Figure 17: Operator F for parameter-lifted programs



of type systems where the edges point in the direction of increasing restrictiveness, i.e. of potentially non-trivial translations. As the syntactic condition that requires the names of all declared functions to be distinct can easily be extended from the level of each `rec` to the global program level, the parameter-lifted type systems can be considered as type system for λ-lifted programs. Variations of traditional SSA occur in the bottom square of the diagram: arguments of Φ-instructions (i.e. ANF function arguments) may or may not be restricted to be variables, and depending on whether a minimal number of Φ-instructions is desired or not, λ-lifting may or may not be required. Calculi $\mathfrak{T}^{\uparrow}_{\rho}$ and $\mathfrak{T}^{\uparrow}_{x,\rho}$ represent languages after the elimination of Φ-instructions, and differ only in the presence of constant load instructions in the compensation code introduced during the elimination. As we will see shortly, the syntactic translation from ANF to GNF discussed in the first part of this paper may be lifted to type transformation along the vertical axis. In particular, the top back right corner, $\mathfrak{T}^{\uparrow}_{x,\rho}$, embodies GNF, for which the coincidence of functional and imperative semantics was proven in [7].

4.5 Proof Transformations

The availability of register information in the types allows us to rephrase the GNF conversion as a translation between type systems. In Figure 18, we give a definition of such a translation from $\mathfrak{T}_x$ to $\mathfrak{T}_{x,\rho}$, defined by three clauses. The algorithm transforms a subderivation $\mathcal{D}$ of shape

$$\text{Call}\,\frac{\mathcal{D}_j : \Gamma \vdash_\Sigma y_j : \sigma_j \quad \Sigma(f) = (\rho_1,\ldots,\rho_n) \to \rho}{\Gamma \vdash_\Sigma f(y_1,\ldots,y_n) : \rho}$$

into a derivation $\Phi_\omega(\mathcal{D})$ where ω is a fresh register. The side conditions on register types in all three rules mirror the syntactic side conditions of the earlier rules G – I to G – III. In rule the second clause, all derivations $\widehat{\mathcal{D}_j}$ are valid since $\Gamma!y_j = \sigma_j \neq \rho_i$ holds for all $j \neq i$. The role of the single additional variable in rule G – III is played by the register ω in the third clause.
More formally, the relationship to algorithm $\mathcal{G}$ may be stated as follows:

Theorem 4.2 *Let $\mathcal{D} : \Gamma \vdash_\Sigma f(y_1,\ldots,y_n) : \rho$ and $\Sigma(f) = (\rho_1,\ldots,\rho_n) \to \rho$. Let α be an allocation for $\mathcal{D}$ and $\omega \notin \{\rho_1,\ldots,\rho_n\} \cup dom\,\Sigma$. Then*

$$(\mathcal{A}_\alpha(\mathcal{D}), [\alpha(\rho_1),\ldots,\alpha(\rho_n)], \alpha(\omega)) \rhd \mathcal{A}_\alpha(\Phi_\omega(\mathcal{D})).$$

$$\text{if } \forall j.\ \rho_j = \sigma_j : \mathcal{D}$$

$$\text{if } \forall j.\ \rho_i \neq \sigma_j \text{ and } x \notin \{y_j | j \neq i\} \cup dom\,\Sigma :$$
$$\frac{\mathcal{D}_i : \Gamma \vdash_\Sigma y_i : \sigma_i \quad \Phi_\omega\left(\dfrac{\forall j \neq i.\ \widehat{\mathcal{D}_j} : \Gamma, x : \rho_i \vdash_\Sigma y_j : \sigma_j \quad \Gamma, x : \rho_i \vdash_\Sigma x : \rho_i}{\Gamma, x : \rho_i \vdash_\Sigma f(y_1, \ldots, y_{i-1}, x, y_{i+1}, \ldots, y_n) : \rho}\right)}{\Gamma \vdash_\Sigma \texttt{let}\ x = y_i\ \texttt{in}\ f(y_1, \ldots, y_{i-1}, x, y_{i+1}, \ldots, y_n) : \rho}$$

$$\text{if } \rho_i \neq \sigma_i \text{ and } \forall k. \exists j. \rho_k = \sigma_j, \text{ and } x \notin \{y_j | j \neq i\} \cup dom\,\Sigma :$$
$$\frac{\mathcal{D}_i : \Gamma \vdash_\Sigma y_i : \sigma_i \quad \Phi_\omega\left(\dfrac{\forall j \text{ s.t. } y_j \neq y_i :\ \widehat{\mathcal{D}_j} : \Gamma, x : \omega \vdash_\Sigma y_j : \sigma_j \quad \Gamma, x : \omega \vdash_\Sigma x : \omega}{\Gamma, x : \omega \vdash_\Sigma f(y_1, \ldots, y_n)[x/y_i] : \rho}\right)}{\Gamma \vdash_\Sigma \texttt{let}\ x = y_i\ \texttt{in}\ f(y_1, \ldots, y_n)[x/y_i] : \rho}$$

Figure 18: Proof transformation Φ_ω for converting $\mathcal{D} \in \mathfrak{T}_x$ to $\Phi_\omega(\mathcal{D}) \in \mathfrak{T}_{x,\rho}$.

Proof See Appendix A.3.

Together with Theorems 2.1 and 4.1, this result establishes operational correctness of type-based register allocation in using Φ.

In imperative compilers, low-level optimisations that interact with register allocation need to be performed after Φ-instructions have been eliminated, and can thus not exploit the SSA structure. The typed setting allows us to phrase such peephole optimisations in a functional setting and to use the ANF structure to justify them. At the level of non-register allocated ANF, Chakravarty et al. rephrased an SSA-based algorithm for performing constant propagation and unreachable code elimination as a functional manipulation, and point out the benefits of such an approach for proving the correctness of the analysis [9]. While we have not yet performed a detailed study of peephole optimisations, it is not difficult to prove the soundness of simple transformations such as

$$\begin{array}{l}\Gamma \vdash_\Sigma \texttt{let}\ x = t_1\ \texttt{in}\ \texttt{let}\ y = t_2\ \texttt{in}\ a : \rho \\ \xrightarrow{t_2 \neq x, t_1 \neq y, x \neq y}\ \Gamma \vdash_\Sigma \texttt{let}\ y = t_2\ \texttt{in}\ \texttt{let}\ x = t_1\ \texttt{in}\ a : \rho\end{array}$$

or of the following optimisation of the code emission function $\mathcal{A}_\alpha(.)$

$$\mathcal{A}_\alpha(\ \text{Let}\ \frac{\Gamma \vdash_\Sigma y : \rho_1 \quad \mathcal{D} : \Gamma, x : \rho_2 \vdash_\Sigma a : \rho}{\Gamma \vdash_\Sigma \texttt{let}\ x = y\ \texttt{in}\ a : \rho}\ ) \xrightarrow{\rho_1 = \rho_2} \mathcal{A}_\alpha(\mathcal{D})$$

which avoids trivial assignments $\texttt{let}\ \alpha(\rho_1) = \alpha(\rho_1)\ \texttt{in}\ \ldots$ arising from instantiation of the Let-rule with $t = y$ and $\rho = \rho_1$.

5 Discussion

In this paper, we demonstrated that the elimination of Φ-instructions from code in SSA form corresponds to a conversion of functional programs from ANF into the more restrictive GNF format. We first introduced the conversion as a purely syntactic translation that combines well-known transformation steps such as λ-lifting with GNF conversion and branch normalisation. We then introduced a family of type systems which model intermediate program representations and are related by proof-transformations. Operational soundness was established using a simple code extraction function that if applied to the most restrictive calculus generates code that can be interpreted functionally or imperatively, without the need for Φ-instructions.

Several authors have recently proposed type-based calculi for register allocation, often using an ANF-like language [1, 2, 24, 28]. While we have restricted our attention to a first-order language with tail-recursive calls, a generalisation to higher-order functions, where caller and callee need to agree on specific register allocation disciplines, is clearly desirable. Indeed, [28], [1] and [2] employ effect systems

to record the impact of more general function calls on registers, and similar annotations are recorded in the types of code pointers in TAL [20].

Ohori's proof-theoretic account of register allocation [24] is based on the sequential sequent calculus (SSC, [23]). Code is represented imperatively, and proof trees are of linear shape. Return instructions are modelled as axioms, and sequential program composition is modelled as the application of syntax-directed proof rules. Structural rules model variable liveness, govern the allocation process, and differentiate between register-based machines and stack-based machines in a style that generalises our context-lookup rules for register variables. Although a relationship with SSA is briefly discussed, no details are given in [24], but [22] explores the relationship between various proof systems and compilation *into* ANF. A more detailed proof-theoretic analysis of our translations would thus complement the work of Ohori, while being notationally closer to functional type systems than the SSC is. Also based on ANF is the type system of Thiemann, where, again, structural rules on contexts model the shadowing of register contents [28].

Agat [1] proposes a type system for a low-level explicitly register-annotated functional form for machines with (finite) register files and (in principle unbounded) stacks. The transition from unallocated to allocated programs is obtained by two operational semantics, the first of which ignores the register annotations and uses a functional interpretation and the second of which models an imperative semantics on register files. In contrast to our setting, program variables do not correspond directly to registers and explicit instructions are introduced that move values between different locations. The soundness result of the type system states that the two semantics coincide for well-typed programs and is proven using a further operational semantics that unifies the two earlier ones. The type system includes effect annotations for closures which ensure that functions expect their arguments in the correct registers and do not interfere with live locations.

Similar to these formal systems, our type system was presented as a mechanism for specifying register allocations. Obtaining allocations amounts to inferring type judgements - a task which we did not address in this paper. Although the system was described using an unstructured set of registers, it can be generalised to include several types of registers (double precision,...), or memory locations, for example by introducing a kinding system. The specific behaviour of different storage locations would then be represented by the availability of kind-specific typing or transformation rules. Thus, the effect of techniques such as spilling could be modelled, although the optimisation task of deciding which intermediates to spill would again be a matter of type inference.

In Morrisett et al.'s Typed Assembly Language, register allocation is performed as the last transformation step [20]. As is the case for our algorithm, function calls are treated individually, but the number of generated moves is slightly higher than that of GNF conversion: a call $f(t_1,\ldots,t_n)$ expands to n move instructions to temporary registers plus n moves to the callee's registers. Furthermore, register names do not α-convert.

Sreedhar et al. [27] propose a technique for eliminating Φ-instructions from SSA code that eliminates more move instructions than earlier mechanisms that rely on post-processing[8, 10]. This algorithm is based on liveness information and a notion of interference between program variables determined by their joint occurrence in a Φ-instruction. The associated congruence classes appear similar to our register types in all type systems stronger than $\mathfrak{T}_\rho$, but a more detailed study is needed to determine how [27]'s algorithm relates to our setting.

Recent contributions to the formal verification of compiler analysis optimisations using theorem prover include [17, 26] at the intermediate level, while [12, 13] verified peephole optimisation steps in PVS.

Although the syntactic view on GNF conversion models more directly the effect of eliminating Φ-instructions in compilers – indeed, one may argue that the violation of α-equivalence is a distinctive feature of low-level languages – the potential to apply (equational) reasoning techniques appears to favour a type-based formulation. In this context, the recent work of Benton [5] appears relevant. This work presented Hoare-style program logics and type systems for reasoning about dependency and con-

stancy information, information flow and dead code elimination in an imperative setting, exploiting the equational theory arising from interpretations of types as partial equivalence relations.

Acknowledgements

I would like to thank Alberto Momigliano for introducing me to proof techniques for coinductive semantics, and Kenneth MacKenzie and Ian Stark for commenting on earlier versions of this paper. I am also grateful to the referees of the COCV workshop, whose suggestions have been very valuable. This work was partially supported by the EPSRC project ReQueST (grant reference EP/C537068/1) and by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project. This paper reflects only the author's views and the Community is not liable for any use that may be made of the information contained therein.

References

[1] Johan Agat. *Type Based Techniques for Covert Channel Elimination and Register Allocation*. PhD thesis, Department of Computing Science, Chalmers University of Technology and Gothenburg University, 2000.

[2] Torben Amtoft and Robert Muller. Inferring annotated types for inter-procedural register allocation with constructor flattening. In *Proceedings of TLDI'03*, SIGPLAN Notices, pages 86–97. ACM Press, January 2003.

[3] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[4] Andrew W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, April 1998.

[5] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of POPL '04*, pages 14–25. ACM, January 2004.

[6] Lennart Beringer. Functional elimination of ϕ-instructions. In *Procceedings of the 5th International Workshop on Compiler Optimization Meets Compiler Verification (COCV'06)*, volume ??(?) of *ENTCS*. Elsevier Science, 2006. To appear.

[7] Lennart Beringer, Kenneth MacKenzie, and Ian Stark. Grail: A functional form for imperative mobile code. In *Proceedings of the 2nd EATCS Workshop on Foundations of Global Computing (FGC'03)*, volume 85(1) of *ENTCS*. Elsevier Science, June 2003.

[8] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software – Practice and Experience*, 28(8):859–881, 1998.

[9] Manuel M. T. Chakravarty, Gabriele Keller, and Patryk Zadarnowski. A functional perspective on SSA optimisation algorithms. In *Procceedings of the 2nd International Workshop on Compiler Optimization Meets Compiler Verification (COCV'03)*, volume 82(2) of *ENTCS*. Elsevier Science, 2003.

[10] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[11] Olivier Danvy and Ulrik P. Schultz. Lambda-dropping: transforming recursive equations into programs with block structure. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial evaluation and semantics-based program manipulation (PEPM '97)*, pages 90–106, New York, NY, USA, 1997. ACM Press.

[12] A. Dold, F.W. von Henke, H. Pfeifer, and H. Rueß. Formal verification of transformations for peephole optimization. In P. Lucas J. Fitzgerald, C.B. Jones, editor, *FME '97: Formal Methods: Their Industrial Application and Strengthened Foundations*, volume 1313 of *LNCS*, pages 459–472. Springer, September 1997.

[13] Axel Dold and Vincent Vialard. Formal verification of a compiler back-end generic checker program. In *Proc. of the Andrei Ershov Third International Conference Perspectives of System Informatics (PSI'99)*, number 1755 in LNCS, pages 470–480. Springer, 2000.

[14] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI'93)*, pages 237–247. ACM, 1993.

[15] Richard A. Kelsey. A correspondence between continuation passing style and static single assignment form. *ACM SIGPLAN Notices*, 30(3):13–22, 1995.

[16] Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. Partial redundancy elimination in SSA form. *ACM Transactions on Programming Languages and Systems*, 21(3):627–676, 1999.

[17] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *Proceedings of the In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2003)*, pages 220–231. ACM, June 2003.

[18] Raymond Lo, Fred Chow, Robert Kennedy, Shin-Ming Liu, and Peng Tu. Register promotion by sparse partial redundancy elimination of loads and stores. *ACM SIGPLAN Notices*, 33(5):26–37, 1998.

[19] Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87:209–220, 1991.

[20] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[21] Steven Muchnick. *Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[22] Atsushi Ohori. A Curry-Howard isomorphism for compilation and program execution. In J.-Y. Girard, editor, *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications (TLCA'99)*, volume 1581 of *LNCS*, pages 280 – 294. Springer, April 1999.

[23] Atsushi Ohori. The Logical Abstract Machine: a Curry-Howard isomorphism for machine code. In Aart Middeldorp and Taisuke Sato, editors, *Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, volume 1722 of *LNCS*, pages 300 – 318. Springer, November 1999.

[24] Atsushi Ohori. Register allocation by proof transformation. In P. Degano, editor, *Proceedings of the 12th European Symposium on Programming (ESOP'03)*, volume 2618 of *LNCS*, pages 399 – 413. Springer, April 2003.

[25] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of POPL '88*, pages 12 – 27. ACM, 1988.

[26] Alexandru Salcianu and Konstantine Arkoudas. Machine-checkable correctness proofs for intra-procedural dataflow analyses. In Jens Knoop, George Necula, and Wolf Zimmermann, editors, *Proceedings of the 4th International Workshop on Compiler Optimization Meets Compiler Verification (COCV'05)*, April 2005.

[27] Vugranam C. Sreedhar, Roy Dz-Ching Ju, David M. Gillies, and Vatsa Santhanam. Translating out of static single assignment form. In *Static Analysis: Proceedings of the 6th International Symposium (SAS'99)*, volume 1694 of *LNCS*, pages 194–210. Springer, September 1999.

[28] Peter Thiemann. Formalizing resource allocation in a compiler. In Xavier Leroy and Atsushi Ohori, editors, *Proceedings of the Second International Workshop on Types in Compilation (TIC'98)*, volume 1473 of *LNCS*, pages 178–193. Springer, March 2003.

A Proofs

The following appendices contain the proofs of the theorems.

A.1 Proof of Theorem 2.1

Proof Induction on the height N of the derivation of $(f(t_1,\ldots,t_n),[x_1,\ldots,x_n],x) \rhd a$.

Case $N = 1$**.** The only possible rule is G-I, hence $a \equiv f(x_1,\ldots,x_n)$ and all $i \in \{1,\ldots,n\}$ satisfy $x_i = t_i$. Hence $a = f(t_1,\ldots,t_n)$ holds and the claim is trivial.

Case $N > 1$**.** There are two cases.

- If the last rule in the derivation of $(f(t_1,\ldots,t_n),[x_1,\ldots,x_n],x) \rhd a$ is G-II, then there are b and i such that $a \equiv \texttt{let}\ x_i = t_i\ \texttt{in}\ b$ and

$$\text{G-II}\ \frac{(f(t_1,\ldots,t_{i-1},x_i,t_{i+1},\ldots,t_n),[x_1,\ldots,x_n],x) \rhd b}{(f(t_1,\ldots,t_n),[x_1,\ldots,x_n],x) \rhd \texttt{let}\ x_i = t_i\ \texttt{in}\ b.}\ \forall j.\ x_i \neq t_j$$

  The derivation of $(f(t_1,\ldots,t_{i-1},x_i,t_{i+1},\ldots,t_n),[x_1,\ldots,x_n],x) \rhd b$ is of height $N-1$, hence, by the induction hypothesis, for all $\mathcal{E}'$ and v' with $\mathcal{E}'(f) = \langle [x_1,\ldots,x_n],\widehat{\mathcal{E}},\widehat{a}\rangle$ we have

$$\mathcal{E}' \vdash f(t_1,\ldots,t_{i-1},x_i,t_{i+1},\ldots,t_n) \Downarrow v' \Leftrightarrow \mathcal{E}' \vdash b \Downarrow v'. \tag{5}$$

  Let $\mathcal{E}$ and v be arbitrary.

  Case $\Rightarrow$**.** The last step in the derivation of $\mathcal{E} \vdash f(t_1,\ldots,t_n) \Downarrow v$ must be

$$\text{CALL}\ \frac{\mathcal{E} \vdash f \Downarrow \langle [x_1,\ldots,x_n],\widehat{\mathcal{E}},\widehat{a}\rangle \quad \widehat{\mathcal{E}}[x_j \mapsto v_j]_{j=1,\ldots,n} \vdash \widehat{a} \Downarrow v}{\mathcal{E} \vdash f(t_1,\ldots,t_n) \Downarrow v}$$

  where $\forall j \in \{1,\ldots,n\}.\mathcal{E} \vdash t_j \Downarrow v_j$. Choose $\mathcal{E}' := \mathcal{E}[x_i \mapsto v_i]$ and $v' := v$. Then the syntactic distinctness of function parameter x_i from function name f yields $\mathcal{E}' \vdash f \Downarrow \langle [x_1,\ldots,x_n],\widehat{\mathcal{E}},\widehat{a}\rangle$, and for all j we have $t_j \neq x_i$ and $\mathcal{E} \vdash t_j \Downarrow v_j$, hence $\mathcal{E}' \vdash t_j \Downarrow v_j$. Therefore, for $(t'_1,\ldots,t'_n) := (t_1,\ldots,t_{i-1},x_i,t_{i+1},\ldots,t_n)$ we obtain $\mathcal{E}' \vdash t'_j \Downarrow v_j$, hence

$$\text{CALL}\ \frac{\mathcal{E}' \vdash f \Downarrow \langle [x_1,\ldots,x_n],\widehat{\mathcal{E}},\widehat{a}\rangle \quad \forall\ j.\ \mathcal{E}' \vdash t'_j \Downarrow v_j \quad \widehat{\mathcal{E}}[x_j \mapsto v_j]_{j=1,\ldots,n} \vdash \widehat{a} \Downarrow v}{\mathcal{E}' \vdash f(t'_1,\ldots,t'_n) \Downarrow v},$$

i.e. $\mathcal{E}' \vdash f(t_1,\ldots,t_{i-1},x_i,t_{i+1},\ldots,t_n) \Downarrow v$. Applying property (5) yields $\mathcal{E}' \vdash b \Downarrow v$, hence

$$\text{LET}\,\frac{\mathcal{E} \vdash t_i \Downarrow v_i \quad \mathcal{E}' \vdash b \Downarrow v}{\mathcal{E} \vdash \texttt{let}\ x_i = t_i\ \texttt{in}\ b \Downarrow v}\,.$$

Case $\Leftarrow$. The last step in the derivation of $\mathcal{E} \vdash a \Downarrow v$ must be

$$\text{LET}\,\frac{\mathcal{E} \vdash t_i \Downarrow w \quad \mathcal{E}[x_i \mapsto w] \vdash b \Downarrow v}{\mathcal{E} \vdash \texttt{let}\ x_i = t_i\ \texttt{in}\ b \Downarrow v}$$

for some w. Choosing $\mathcal{E}' := \mathcal{E}[x_i \mapsto w]$ and $v' := v$ in property (5) yields

$$\mathcal{E}' \vdash f(t_1,\ldots,t_{i-1},x_i,t_{i+1},\ldots,t_n) \Downarrow v.$$

For the same reasons as in the above case we also have $\mathcal{E}' \vdash f \Downarrow \langle [x_1,\ldots,x_n], \widehat{\mathcal{E}}, \widehat{a} \rangle$, hence for $(t'_1,\ldots,t'_n) := (t_1,\ldots,t_{i-1},x_i,t_{i+1},\ldots,t_n)$ we have

$$\text{CALL}\,\frac{\forall\ j.\ \mathcal{E}' \vdash t'_j \Downarrow v_j \qquad \widehat{\mathcal{E}}[x_j \mapsto v_j]_{j=1,\ldots,n} \vdash \widehat{a} \Downarrow v}{\mathcal{E}' \vdash f(t'_1,\ldots,t'_n) \Downarrow v}$$

for some $v_1,\ldots,v_n$. For all j with $j \neq i$ we have $t'_j = t_j \neq x_i$, hence $\mathcal{E} \vdash t_j \Downarrow v_j$. Also, $v_i = w$ holds, hence $\mathcal{E} \vdash t_i \Downarrow v_i$ and therefore

$$\text{CALL}\,\frac{\forall\ j.\ \mathcal{E} \vdash t_j \Downarrow v_j \qquad \widehat{\mathcal{E}}[x_j \mapsto v_j]_{j=1,\ldots,n} \vdash \widehat{a} \Downarrow v}{\mathcal{E} \vdash f(t_1,\ldots,t_n) \Downarrow v}\,.$$

- If the last rule in the derivation of $(f(t_1,\ldots,t_n),[x_1,\ldots,x_n],x) \rhd a$ is G-III, then there are b and i such that $a \equiv \texttt{let}\ x = x_i\ \texttt{in}\ b$ and

$$\text{G-III}\,\frac{(f(t_1,\ldots,t_n)[x/x_i],x_1,\ldots,x_n,x) \rhd b}{(f(t_1,\ldots,t_n),x_1,\ldots,x_n,x) \rhd \texttt{let}\ x = x_i\ \texttt{in}\ b}\ \begin{cases} x_i \neq t_i \\ \forall k.\exists j.\ x_k = t_j \end{cases}$$

Hence, there is at least one $\nu \in \{1,\ldots,n\} \setminus \{i\}$ with

$$x_i = t_\nu. \tag{6}$$

The derivation of $f(t_1,\ldots,t_n)[x/x_i],[x_1,\ldots,x_n],x) \rhd b$ is of height $N-1$, hence, by the induction hypothesis, for all $\mathcal{E}'$ and v' with $\mathcal{E}'(f) = \langle [x_1,\ldots,x_n], \widehat{\mathcal{E}}, \widehat{a} \rangle$ we have

$$\mathcal{E}' \vdash f(t_1,\ldots,t_n)[x/x_i] \Downarrow v' \Leftrightarrow \mathcal{E}' \vdash b \Downarrow v'. \tag{7}$$

It is also not difficult to see that the two side conditions of G-III, together with $x \notin \{x_1,\ldots,x_n\}$ and the distinctness of the x_i, imply

$$x \notin \{t_1,\ldots,t_n\}. \tag{8}$$

as the sets $\{x_1,\ldots,x_n\}$ and $\{t_1,\ldots,t_n\}$ are identical.
Let $\mathcal{E}$ and v be arbitrary.

Case $\Rightarrow$. The last step in the derivation of $\mathcal{E} \vdash f(t_1,\ldots,t_n) \Downarrow v$ must be

$$\text{CALL}\,\frac{\mathcal{E} \vdash f \Downarrow \langle [x_1,\ldots,x_n], \widehat{\mathcal{E}}, \widehat{a} \rangle \quad \forall\ j.\ \mathcal{E} \vdash t_j \Downarrow v_j \quad \widehat{\mathcal{E}}[x_j \mapsto v_j]_{j=1,\ldots,n} \vdash \widehat{a} \Downarrow v}{\mathcal{E} \vdash f(t_1,\ldots,t_n) \Downarrow v}\,.$$

Choose $\mathcal{E}' := \mathcal{E}[x \mapsto v_\mathrm{v}]$ and $v' := v$. From (6) and $\forall\ j.\ \mathcal{E} \vdash t_j \Downarrow v_j$ we get

$$\mathcal{E}(x_i) = v_\mathrm{v}, \tag{9}$$

hence for all j with $v_j = x_i$ we have $v_j = v_\mathrm{v}$ and thus $\mathcal{E}' \vdash t_j[x/x_i] \Downarrow v_j$, since $t_j[x/x_i] = x$. For all j with $t_j \neq x_i$ we have $t_j[x/x_i] = t_j$, hence $\mathcal{E}' \vdash t_j[x/x_i] \Downarrow v_j$, as $t_j \neq x$ holds by (8). Thus, for all j we have $\mathcal{E}' \vdash t_j[x/x_i] \Downarrow v_j$, hence

$$\mathrm{CALL}\,\frac{\mathcal{E}' \vdash f \Downarrow \langle [x_1,\ldots,x_n], \widehat{\mathcal{E}}, \widehat{a}\rangle \quad \widehat{\mathcal{E}}[x_j \mapsto v_j]_{j=1,\ldots,n} \vdash \widehat{a} \Downarrow v}{\mathcal{E}' \vdash f(t_1,\ldots,t_n)[x/x_i] \Downarrow v}\,.$$

Applying the induction hypothesis (7) yields $\mathcal{E}' \vdash b \Downarrow v$, i.e. $\mathcal{E}[x \mapsto v_\mathrm{v}] \vdash b \Downarrow v$ by the definition of $\mathcal{E}'$, hence, by (9),

$$\mathrm{LET}\,\frac{\mathcal{E} \vdash x_i \Downarrow v_\mathrm{v} \quad \mathcal{E}[x \mapsto v_\mathrm{v}] \vdash b \Downarrow v}{\mathcal{E} \vdash \mathtt{let}\ x = x_i\ \mathtt{in}\ b \Downarrow v}$$

as required.

Case $\Leftarrow$. The last step in the derivation of $\mathcal{E} \vdash a \Downarrow v$ must be

$$\mathrm{LET}\,\frac{\mathcal{E} \vdash x_i \Downarrow w \quad \mathcal{E}[x \mapsto w] \vdash b \Downarrow v}{\mathcal{E} \vdash \mathtt{let}\ x = x_i\ \mathtt{in}\ b \Downarrow v}$$

for some w. Choosing $\mathcal{E}' := \mathcal{E}[x \mapsto w]$ and $v' := v$ in the induction hypothesis (7) yields

$$\mathcal{E}[x \mapsto w] \vdash f(t_1,\ldots,t_n)[x/x_i] \Downarrow v,$$

hence

$$\mathrm{CALL}\,\frac{\mathcal{E}' \vdash f \Downarrow \langle [x_1,\ldots,x_n], \widehat{\mathcal{E}}, \widehat{a}\rangle \quad \forall\ j.\ \mathcal{E}' \vdash t_j[x/x_i] \Downarrow v_j \quad \widehat{\mathcal{E}}[x_j \mapsto v_j]_{j=1,\ldots,n} \vdash \widehat{a} \Downarrow v}{\mathcal{E}' \vdash f(t_1,\ldots,t_n)[x/x_i] \Downarrow v}$$

for some $v_1,\ldots,v_n$.
For all j with $t_j = x_i$ we first have $t_j[x/x_i] = x$ (in particular, we have $t_\mathrm{v}[x/x_i] = x$), and therefore $v_j = w$. On the other hand, the first subtree in the application of rule LET above yields $\mathcal{E}(x_i) = w$, hence these j also fulfill $\mathcal{E} \vdash t_j \Downarrow w$. Combining these two facts yields

$$\mathcal{E} \vdash t_j \Downarrow v_j \text{ for all } j \text{ with } t_j = x_i. \tag{10}$$

For the j with $t_j \neq x_i$, we observe that $t_j[x/x_i] \neq x$ holds, since $t_j \neq x$ follows from (8). Hence the definition of v_j in the j-th subtree of the application of rule CALL yields

$$\mathcal{E} \vdash t_j \Downarrow v_j \text{ for all } j \text{ with } t_j \neq x_i. \tag{11}$$

Combining (10) and (11) yields $\mathcal{E} \vdash t_j \Downarrow v_j$ for all j, hence

$$\mathrm{CALL}\,\frac{\mathcal{E} \vdash f \Downarrow \langle [x_1,\ldots,x_n], \widehat{\mathcal{E}}, \widehat{a}\rangle \quad \forall\ j.\ \mathcal{E} \vdash t_j \Downarrow v_j \qquad \widehat{\mathcal{E}}[x_j \mapsto v_j]_{j=1,\ldots,n} \vdash \widehat{a} \Downarrow v}{\mathcal{E} \vdash f(t_1,\ldots,t_n) \Downarrow v}\,.$$

A.2 Proof of Theorem 4.1

Proof Both items are proven separately, by induction on the operational semantics.

- Induction on (the height of) $\mathcal{E} \vdash a \Downarrow v$.

  Case CONST. For $a \equiv c$, the derivation of $\mathcal{E} \vdash a \Downarrow v$ implies $v = c$, and from $\mathcal{D} : \Gamma \vdash_\Sigma a : \tau$ we deduce $\tau = \rho$ and $\mathcal{A}_\alpha(\mathcal{D}) = a = c$ for some ρ, hence $\mathcal{F} \vdash \mathcal{A}_\alpha(\mathcal{D}) \Downarrow v$ follows from rule CONST.

  Case REC. For $a \equiv \mathtt{rec}\ [x_i\ x_1^i \ldots x_{n_i}^i = a_i]_{i=1,\ldots,n}\ \mathtt{in}\ \widehat{a}$, the derivation of $\mathcal{E} \vdash a \Downarrow v$ has shape

$$\text{REC}\,\frac{\widehat{\mathcal{E}} \vdash \widehat{a} \Downarrow v \quad \widehat{\mathcal{E}} = \mathcal{E}[x_i \mapsto \langle [x_1^i, \ldots, x_{n_i}^i], \widehat{\mathcal{E}}, a_i \rangle]_{i=1,\ldots,n}}{\mathcal{E} \vdash a \Downarrow v},$$

  $\mathcal{D}$ has shape

$$\text{Rec}\,\frac{\begin{array}{l}\mathcal{D}_0 : \Gamma \vdash_\Pi \widehat{a} : \rho \\ \forall i.\ \mathcal{D}_i : \Gamma, x_1^i : \rho_1^i, \ldots, x_{n_i}^i : \rho_{n_i}^i \vdash_\Pi a_i : \rho_i \\ \forall i\ j\ k.\ j \neq k \Rightarrow \rho_j^i \neq \rho_k^i \\ \forall i.\ x_i \notin dom\,\Gamma \cup dom\,\Sigma \\ \forall i\ j.\ x_j^i \notin dom\,\Pi \\ \Pi = \Sigma[x_i \mapsto (\rho_1^i, \ldots, \rho_{n_i}^i) \to \rho_i]_{i=1,\ldots,n}\end{array}}{\Gamma \vdash_\Sigma a : \rho},$$

  and $\mathcal{A}_\alpha(\mathcal{D}) = \mathtt{rec}\ [x_i\ \alpha(\rho_1^i) \ldots \alpha(\rho_{n_i}^i) = \mathcal{A}_\alpha(\mathcal{D}_i)]_{i=1,\ldots,n}\ \mathtt{in}\ \mathcal{A}_\alpha(\mathcal{D}_0)$. We will show that $\widehat{\mathcal{E}} \approx^{(\Gamma,\Pi)}_{fv(\widehat{a})} \widehat{\mathcal{F}}$ holds, where

$$\widehat{\mathcal{F}} = \mathcal{F}[x_i \mapsto \langle [\alpha(\rho_1^i), \ldots, \alpha(\rho_{n_i}^i)], \widehat{\mathcal{F}}, \mathcal{A}_\alpha(\mathcal{D}_i) \rangle]_{i=1,\ldots,n}.$$

  This will make the induction hypothesis applicable to $\widehat{\mathcal{E}} \vdash \widehat{a} \Downarrow v$, so $\mathcal{D}_0$ will guarantee the existence of a w with $\widehat{\mathcal{F}} \vdash \mathcal{A}_\alpha(\mathcal{D}_0) \Downarrow w$ and $v \sim_\rho w$, i.e. $v = w = c$ for some c. From this the claim will follow using

$$\text{REC}\,\frac{\widehat{\mathcal{F}} \vdash \mathcal{A}_\alpha(\mathcal{D}_0) \Downarrow w \quad \widehat{\mathcal{F}} = \mathcal{F}[x_i \mapsto \langle [\alpha(\rho_1^i), \ldots, \alpha(\rho_{n_i}^i)], \widehat{\mathcal{F}}, \mathcal{A}_\alpha(\mathcal{D}_i) \rangle]_{i=1,\ldots,n}}{\mathcal{F} \vdash \mathcal{A}_\alpha(\mathcal{D}) \Downarrow w.}$$

  For showing $\widehat{\mathcal{E}} \approx^{(\Gamma,\Pi)}_{fv(\widehat{a})} \widehat{\mathcal{F}}$, we observe that $fv(\widehat{a}) \subseteq fv(a) \cup \{x_1, \ldots, x_n\}$ holds, and that

  i) $fv(\widehat{a}) \subseteq dom\,\widehat{\mathcal{E}} \cap (dom\,\Gamma \cup dom\,\Pi)$ holds: from $\mathcal{E} \approx^{(\Gamma,\Sigma)}_{fv(a)} \mathcal{F}$ we deduce $fv(a) \subseteq dom\,\mathcal{E} \cap (dom\,\Gamma \cup dom\,\Sigma)$, hence

$$\begin{array}{rcl} fv(\widehat{a}) & \subseteq & fv(a) \cup \{x_1, \ldots, x_n\} \\ & \subseteq & (dom\,\mathcal{E} \cap (dom\,\Gamma \cup dom\,\Sigma)) \cup \{x_1, \ldots, x_n\} \\ & \subseteq & (dom\,\mathcal{E} \cup \{x_1, \ldots, x_n\}) \cap (dom\,\Gamma \cup dom\,\Sigma \cup \{x_1, \ldots, x_n\}) \\ & = & (dom\,\widehat{\mathcal{E}}) \cap (dom\,\Gamma \cup dom\,\Pi). \end{array}$$

  ii) for all $x \in fv(\widehat{a}) \cap dom\,\Gamma$, $\widehat{\mathcal{E}}(x) \sim_{\Gamma!x} \widehat{\mathcal{F}}(y)$ holds, where $y = \alpha(\Gamma!x)$: we have
  - $x \neq x_i$ for all $i \in \{1, \ldots, n\}$ due to the side condition $x_i \notin dom\,\Gamma$, hence $\widehat{\mathcal{E}}(x) = \mathcal{E}(x)$
  - $y \neq x_i$ for all $i \in \{1, \ldots, n\}$ (as $cod\,\alpha \cap dom\,\Pi = \emptyset$ follows from the fact that α is a register allocation for $\mathcal{D}$ and thus for $\mathcal{D}_0$), thus $\widehat{\mathcal{F}}(y) = \mathcal{F}(y)$

– $x \in fv(a) \cap dom\,\Gamma$ (since $x \neq x_i$ for all i), so item ii) of the definition of $\mathcal{E} \approx^{(\Gamma,\Sigma)}_{fv(a)} \mathcal{F}$ yields $\mathcal{E}(x) \sim_{\Gamma!x} \mathcal{F}(\alpha(y))$.

iii) for all $x \in fv(\widehat{a}) \cap dom\,\Pi$, $\widehat{\mathcal{E}}(x) \sim_{\Pi(x)} \widehat{\mathcal{F}}(x)$ holds: if $x \neq x_i$ for all $i \in \{1,\ldots,n\}$ then $x \in fv(a) \cap dom\,\Sigma$, $\widehat{\mathcal{E}}(x) = \mathcal{E}(x)$, $\widehat{\mathcal{F}}(x) = \mathcal{F}(x)$ and $\Pi(x) = \Sigma(x)$, so $\widehat{\mathcal{E}}(x) \sim_{\Pi(x)} \widehat{\mathcal{F}}(x)$ follows from $\mathcal{E}(x) \sim_{\Sigma(x)} \mathcal{F}(x)$ (which holds because to $\mathcal{E} \approx^{(\Gamma,\Sigma)}_{fv(a)} \mathcal{F}$). Otherwise, $x = x_i$ holds for some (unique) $i \in \{1,\ldots,n\}$, and we obtain

$$\begin{aligned}
\widehat{\mathcal{E}}(x) &= \langle [x^i_1,\ldots,x^i_{n_i}], \widehat{\mathcal{E}}, a_i \rangle, \\
\widehat{\mathcal{F}}(x) &= \langle [\alpha(\rho^i_1),\ldots,\alpha(\rho^i_{n_i})], \widehat{\mathcal{F}}, \mathcal{A}_\alpha(\mathcal{D}_i) \rangle, \text{ and} \\
\Pi(x) &= (\rho^i_1,\ldots,\rho^i_{n_i}) \to \rho_i.
\end{aligned}$$

From this we deduce $\widehat{\mathcal{E}}(x) \sim_{\Pi(x)} \widehat{\mathcal{F}}(x)$ as follows. For the set

$$Q := Q^{max} \cup \left\{ \begin{array}{l} (\ \langle [x^j_1,\ldots,x^j_{n_j}], \widehat{\mathcal{E}}, a_j \rangle, \\ \ (\rho^j_1,\ldots,\rho^j_{n_j}) \to \rho_j, \\ \ \langle [\alpha(\rho^j_1),\ldots,\alpha(\rho^j_{n_j})], \widehat{\mathcal{F}}, \mathcal{A}_\alpha(\mathcal{D}_j) \rangle\ ) \end{array} \middle| \ 1 \leq j \leq n \right\}$$

we will show $Q \subseteq F(Q)$, since this property implies $Q \subseteq Q^{max}$, from which $\widehat{\mathcal{E}}(x) \sim_{\Pi(x)} \widehat{\mathcal{F}}(x)$ follows using $(\widehat{\mathcal{E}}(x), \Pi(x), \widehat{\mathcal{F}}(x)) \in Q$. So let $q := (\nu', \tau, w') \in Q$.
If $q \in Q^{max}$, then from $Q^{max} \subseteq Q$ and the monotonicity of F we deduce $F(Q^{max}) \subseteq F(Q)$, and the fixed point property of Q^{max} yields $Q^{max} = F(Q^{max})$, hence

$$q \in Q^{max} = F(Q^{max}) \subseteq F(Q).$$

Otherwise, there is a (unique) $j \in \{1,\ldots,n\}$ such that

$$\begin{aligned}
\nu' &= \langle [x^j_1,\ldots,x^j_{n_j}], \widehat{\mathcal{E}}, a_j \rangle, \\
w' &= \langle [\alpha(\rho^j_1),\ldots,\alpha(\rho^j_{n_j})], \widehat{\mathcal{F}}, \mathcal{A}_\alpha(\mathcal{D}_j) \rangle \text{ and} \\
\tau &= (\rho^j_1,\ldots,\rho^j_{n_j}) \to \rho_j.
\end{aligned}$$

Aiming to establish the conditions of operator F, we first notice that elements of the list $[x^j_1,\ldots,x^j_{n_j}]$ are distinct by the syntactic conditions on programs introduced in the first section. Second, the elements of $[\alpha(\rho^j_1),\ldots,\alpha(\rho^j_{n_j})]$ have the form $\alpha(.)$ mandated by the definition of operator F, and they are distinct by the third side condition of rule Rec and the fact that α is an allocation for $\mathcal{D}$ and thus for $\mathcal{D}_j$. Finally, for $\Delta := \Gamma, x^j_1 : \rho^j_1, \ldots, x^j_{n_j} : \rho^j_{n_j}$ and arbitrary $y \in fv(a_j) \setminus \{x^j_1,\ldots,x^j_{n_j}\}$ the four items in the definition of F are satisfied:

I) $\mathcal{D}_j : \Delta \vdash_\Pi a_j : \rho_j$ by the second side condition of rule Rec. Well-formedness of $\mathcal{D}_j$ follows from that of $\mathcal{D}$, and so does the fact that α is an allocation for $\mathcal{D}_j$. Also, the third component of w' equals $\mathcal{A}_\alpha(\mathcal{D}_j)$.

II) $y \in dom\,\widehat{\mathcal{E}} \cap (dom\,\Delta \cup dom\,\Pi)$ holds: from Lemma 1 and I) we obtain $y \in dom\,\Delta \cup dom\,\Pi$, and assumption $\mathcal{E} \approx^{(\Gamma,\Sigma)}_{fv(a)} \mathcal{F}$ yields

$$\begin{aligned}
fv(a_j) &\subseteq fv(a) \cup \{x_1,\ldots,x_n\} \cup \{x^j_1,\ldots,x^j_{n_j}\} \\
&\subseteq (dom\,\mathcal{E} \cap (dom\,\Gamma \cup dom\,\Sigma)) \cup \{x_1,\ldots,x_n\} \cup \{x^j_1,\ldots,x^j_{n_j}\} \\
&\subseteq dom\,\mathcal{E} \cup \{x_1,\ldots,x_n\} \cup \{x^j_1,\ldots,x^j_{n_j}\},
\end{aligned}$$

hence $y \in dom\ \mathcal{E} \cup \{x_1,\ldots,x_n\} = dom\ \widehat{\mathcal{E}}$.

III) if $y \in dom\ \Delta$ then

$$\alpha(\Delta!y) \in dom\ \widehat{\mathcal{F}} \setminus \{\alpha(\rho_1^j),\ldots,\alpha(\rho_{n_j}^j)\}$$

and

$$(\widehat{\mathcal{E}}(y), \Delta!y, \widehat{\mathcal{F}}(\alpha(\Delta!y))) \in Q$$

hold: from $y \in dom\ \Delta \setminus \{x_1^j,\ldots,x_{n_j}^j\}$ we deduce $y \in dom\ \Gamma$, and in particular, $\Delta!y = \Gamma!y \neq \rho_k^j$ for all $k \in \{1,\ldots,n_j\}$, i.e.

$$\alpha(\Delta!y) \notin \{\alpha(\rho_1^j),\ldots,\alpha(\rho_{n_j}^j)\}.$$

From $y \in dom\ \Gamma$ and the fourth side condition of rule Rec we obtain $y \neq x_i$ for all $i \in \{1,\ldots,n\}$, hence

$$\widehat{\mathcal{E}}(y) = \mathcal{E}(y) \text{ and } y \in fv(a).$$

From $y \in fv(a)$ we deduce $y \in dom\ \mathcal{E}$ using the first item of the definition of $\mathcal{E} \approx^{(\Gamma,\Sigma)}_{fv(a)} \mathcal{F}$, hence the second item of the same definition (together with $y \in dom\ \Gamma$) yields

$$\alpha(\Gamma!y) \in dom\ \mathcal{F} \ \subseteq dom\ \widehat{\mathcal{F}} \text{ and } \mathcal{E}(y) \sim_{\Gamma!y} \mathcal{F}(\alpha(\Gamma!y)).$$

From the condition $cod\ \alpha \cap dom\ \Pi = \emptyset$ (satisfied as α is a register allocation for $\mathcal{D}$ and $\mathcal{D}_j$ a subderivation of $\mathcal{D}$) we obtain $\alpha(\Gamma!y) \neq x_i$ for all $i \in \{1,\ldots,n\}$, thus

$$\widehat{\mathcal{F}}(\alpha(\Delta!y)) = \widehat{\mathcal{F}}(\alpha(\Gamma!y)) = \mathcal{F}(\alpha(\Gamma!y)).$$

Combining all this yields $\alpha(\Gamma!y) \in dom\ \widehat{\mathcal{F}} \setminus \{\alpha(\rho_1^j),\ldots,\alpha(\rho_{n_j}^j)\}$ and $\widehat{\mathcal{E}}(y) \sim_{\Delta!y} \widehat{\mathcal{F}}(\alpha(\Delta!y))$, i.e.

$$(\widehat{\mathcal{E}}(y), \Delta!y, \widehat{\mathcal{F}}(\alpha(\Delta!y))) \in Q^{max} \subseteq Q.$$

IV) if $y \in dom\ \Pi$ then $(\widehat{\mathcal{E}}(y), \Pi(y), \widehat{\mathcal{F}}(y)) \in Q$ holds: If $y = x_k$ for some (unique) $k \in \{1,\ldots,n\}$, then

$$\begin{aligned}
\widehat{\mathcal{E}}(y) &= \langle [x_1^k,\ldots,x_{n_k}^k], \widehat{\mathcal{E}}, a_k \rangle \\
\Pi(y) &= (\rho_1^k,\ldots,\rho_{n_k}^k) \to \rho_k \\
\widehat{\mathcal{F}}(y) &= \langle [\alpha(\rho_1^k),\ldots,\alpha(\rho_{n_k}^k)], \widehat{\mathcal{F}}, \mathcal{A}_\alpha(\mathcal{D}_k) \rangle
\end{aligned}$$

so $(\widehat{\mathcal{E}}(y), \Pi(y), \widehat{\mathcal{F}}(y)) \in Q$. Otherwise, we have $y \in dom\ \Sigma$ and $y \in fv(a)$, so the third item of the definition of $\mathcal{E} \approx^{(\Gamma,\Sigma)}_{fv(a)} \mathcal{F}$ implies

$$\mathcal{E}(y) \sim_{\Sigma(y)} \mathcal{F}(y).$$

Also, $y \notin \{x_1,\ldots,x_n\}$ implies $\widehat{\mathcal{E}}(y) = \mathcal{E}(y)$, $\widehat{\mathcal{F}}(y) = \mathcal{F}(y)$ and $\Pi(y) = \Sigma(y)$, and we obtain

$$\widehat{\mathcal{E}}(y) \sim_{\Pi(y)} \widehat{\mathcal{F}}(y),$$

i.e. $(\widehat{\mathcal{E}}(y), \Pi(y), \widehat{\mathcal{F}}(y)) \in Q^{max} \subseteq Q$.

Case VAR. For $a \equiv x$ the derivation of $\mathcal{E} \vdash a \Downarrow v$ yields $x \in dom\ \mathcal{E}$ with $v = \mathcal{E}(x)$, and $\mathcal{D}$ yields $\Gamma!x = \rho$ for some ρ, i.e. $x \in dom\ \Gamma$, and $\mathcal{A}_\alpha(\mathcal{D}) = \alpha(\rho)$. From $x \in fv(a)$ and $\mathcal{E} \approx^{fv(a)}_{(\Gamma,\Sigma)} \mathcal{F}$ we obtain $\mathcal{E}(x) \sim_\rho \mathcal{F}(\alpha(\rho))$, i.e. $\mathcal{F}(\alpha(\rho)) = v = c$ for some c, and rule VAR implies $\mathcal{F} \vdash \alpha(\rho) \Downarrow c$.

Case CALL. For $a \equiv x(t_1,\ldots,t_n)$, the derivation of $\mathcal{E} \vdash a \Downarrow v$ has shape

$$\text{CALL}\,\frac{\mathcal{E} \vdash x \Downarrow \langle [y_1,\ldots,y_n], \widehat{\mathcal{E}}, \widehat{a}\rangle \quad \forall i.\ \mathcal{E} \vdash t_i \Downarrow v_i \quad \widehat{\mathcal{E}}[y_i \mapsto v_i]_{i=1,\ldots,n} \vdash \widehat{a} \Downarrow v}{\mathcal{E} \vdash x(t_1,\ldots,t_n) \Downarrow v}$$

for some (unique) $y_1,\ldots,y_n, \widehat{\mathcal{E}}, \widehat{a}$ and $v_1,\ldots,v_n$, i.e.

$$\mathcal{E}(x) = \langle [y_1,\ldots,y_n], \widehat{\mathcal{E}}, \widehat{a}\rangle.$$

Derivation $\mathcal{D}$ has shape

$$\text{Call}\,\frac{\Sigma(x) = (\rho_1,\ldots,\rho_n) \to \rho \qquad \forall i \in \{1,\ldots,n\}.\mathcal{D}_i : \Gamma \vdash_\Sigma t_i : \rho_i'}{\Gamma \vdash_\Sigma x(t_1,\ldots,t_n) : \rho},$$

hence $\mathcal{A}_\alpha(\mathcal{D}) = x(\mathcal{A}_\alpha(\mathcal{D}_1),\ldots,\mathcal{A}_\alpha(\mathcal{D}_n))$. From $\mathcal{E} \approx^{(\Gamma,\Sigma)}_{fv(a)} \mathcal{F}$ and $fv(t_i) \subseteq fv(a)$ we obtain $\mathcal{E} \approx^{(\Gamma,\Sigma)}_{fv(t_i)} \mathcal{F}$, so we can apply the induction hypothesis to $\mathcal{E} \vdash t_i \Downarrow v_i$, so $\mathcal{D}_i$ guarantees the existence of some w_i such that

$$\mathcal{F} \vdash \mathcal{A}_\alpha(\mathcal{D}_i) \Downarrow w_i$$

and $v_i \sim_{\rho_i'} w_i$. Thus, $v_i = w_i = c_i$ for some c_i and all $i \in \{1,\ldots,n\}$.

From $x \in fv(a) \cap dom\,\Sigma$ and the definition of $\mathcal{E} \approx^{(\Gamma,\Sigma)}_{fv(a)} \mathcal{F}$ we obtain $\mathcal{E}(x) \sim_{\Sigma(x)} \mathcal{F}(x)$, hence

$$\mathcal{F}(x) = \langle [\alpha(\rho_1),\ldots,\alpha(\rho_n)], \widehat{\mathcal{F}}, b\rangle$$

for some unique $\widehat{\mathcal{F}}$ and b, the $\alpha(\rho_i)$ are distinct, the y_i are distinct, and there are Γ' and Π such that for $\Delta := \Gamma', y_1 : \rho_1,\ldots,y_n : \rho_n$ and $z \in fv(\widehat{a}) \setminus \{y_1,\ldots,y_n\}$ we have

I) $\mathcal{D}^* : \Delta \vdash_\Pi \widehat{a} : \rho$ is a well-formed derivation, α is an allocation for $\mathcal{D}^*$, and $b = \mathcal{A}_\alpha(\mathcal{D})^*$

II) $z \in dom\,\widehat{\mathcal{E}} \cap (dom\,\Delta \cup dom\,\Pi)$

III) if $z \in dom\,\Delta$ then $\alpha(\Delta!z) \in dom\,\widehat{\mathcal{F}} \setminus \{\alpha(\rho_1),\ldots,\alpha(\rho_n)\}$ and

$$(\widehat{\mathcal{E}}(z), \Delta!z, \widehat{\mathcal{F}}(\alpha(\Delta!z))) \in Q^{max}, \text{ i.e. } \widehat{\mathcal{E}}(z) \sim_{\Delta!z} \widehat{\mathcal{F}}(\alpha(\Delta!z))$$

IV) if $z \in dom\,\Pi$ then $(\widehat{\mathcal{E}}(z), \Pi(z), \widehat{\mathcal{F}}(z)) \in Q^{max}$, i.e. $\widehat{\mathcal{E}}(z) \sim_{\Pi(z)} \widehat{\mathcal{F}}(z)$.

We will show that $\widehat{\mathcal{E}}[y_i \mapsto c_i]_{i=1,\ldots,n} \approx^{(\Delta,\Pi)}_{fv(\widehat{a})} \widehat{\mathcal{F}}[\alpha(\rho_i) \mapsto c_i]_{i=1,\ldots,n}$ holds. This will make the induction hypothesis applicable to $\widehat{\mathcal{E}}[y_i \mapsto c_i]_{i=1,\ldots,n} \vdash \widehat{a} \Downarrow v$ and thus guarantee the existence of a w with $\widehat{\mathcal{F}}[\alpha(\rho_i) \mapsto c_i]_{i=1,\ldots,n} \vdash \mathcal{A}_\alpha(\mathcal{D}^*) \Downarrow w$, i.e.

$$\text{CALL}\,\frac{\begin{array}{c}\mathcal{F} \vdash x \Downarrow \langle [\alpha(\rho_1),\ldots,\alpha(\rho_n)], \widehat{\mathcal{F}}, b\rangle \\ \forall i.\ \mathcal{F} \vdash \mathcal{A}_\alpha(\mathcal{D}_i) \Downarrow c_i \quad \widehat{\mathcal{F}}[\alpha(\rho_i) \mapsto c_i]_{i=1,\ldots,n} \vdash b \Downarrow v\end{array}}{\mathcal{F} \vdash x(\mathcal{A}_\alpha(\mathcal{D}_1),\ldots,\mathcal{A}_\alpha(\mathcal{D}_n)) \Downarrow v}$$

and $v \sim_\rho w$. Indeed, we have $\widehat{\mathcal{E}}[y_i \mapsto c_i]_{i=1,\ldots,n} \approx^{(\Delta,\Pi)}_{fv(\widehat{a})} \widehat{\mathcal{F}}[\alpha(\rho_i) \mapsto c_i]_{i=1,\ldots,n}$:

i) We have $fv(\widehat{a}) \subseteq dom\,\widehat{\mathcal{E}}[y_i \mapsto c_i]_{i=1,\ldots,n} \cap (dom\,\Delta \cup dom\,\Pi)$: if $y \in fv(\widehat{a}) \setminus \{y_1,\ldots,y_n\}$ we have $y \in dom\,\widehat{\mathcal{E}} \subseteq dom\,\widehat{\mathcal{E}}[y_i \mapsto c_i]_{i=1,\ldots,n}$ and $y \in dom\,\Delta \cup dom\,\Pi$ by item II) above. If $y = y_k$ for some $k \in \{1,\ldots,n\}$ then $y \in dom\,\widehat{\mathcal{E}}[y_i \mapsto c_i]_{i=1,\ldots,n}$ holds trivially, and $y \in dom\,\Delta$ follows from the distinctness of the y_i and the ρ_i

ii) For $y \in fv(\widehat{a}) \cap dom\,\Delta$, $\widehat{\mathcal{E}}[y_i \mapsto c_i]_{i=1,\ldots,n}(y) \sim_{\Delta!y} \widehat{\mathcal{F}}\,[\alpha(\rho_i) \mapsto c_i]_{i=1,\ldots,n}(\Delta!y)$ holds: if $y \in fv(\widehat{a}) \setminus \{y_1,\ldots,y_n\}$ then

$$\widehat{\mathcal{E}}[y_i \mapsto c_i]_{i=1,\ldots,n}(y) = \widehat{\mathcal{E}}(y)$$

holds, and from $y \in dom\,\Delta$ we obtain $\Delta!y \neq \rho_i$ for all $i \in \{1,\ldots,n\}$, thus $\alpha(\Delta!y) \neq \alpha(\rho_i)$, and therefore

$$\widehat{\mathcal{F}}\,[\alpha(\rho_i) \mapsto c_i]_{i=1,\ldots,n}(\alpha(\Delta!y)) = \widehat{\mathcal{F}}\,(\alpha(\Delta!y)).$$

From item III) above we also have $\alpha(\Delta!y) \in dom\,\widehat{\mathcal{F}} \setminus \{\alpha(\rho_1),\ldots,\alpha(\rho_n)\}$ and

$$\widehat{\mathcal{E}}(y) \sim_{\Delta!y} \widehat{\mathcal{F}}\,(\alpha(\Delta!y)),$$

hence $\widehat{\mathcal{E}}[y_i \mapsto c_i]_{i=1,\ldots,n}(y) \sim_{\Delta!y} \widehat{\mathcal{F}}\,[\alpha(\rho_i) \mapsto c_i]_{i=1,\ldots,n}(\alpha(\Delta!y))$ as required. If $y = y_k$ for some (unique) $k \in \{1,\ldots,n\}$, then

$$\begin{array}{rcl} \widehat{\mathcal{E}}[y_i \mapsto c_i]_{i=1,\ldots,n}(y) & = & c_k, \\ \Delta!y & = & \rho_k \text{ and} \\ \widehat{\mathcal{F}}\,[\alpha(\rho_i) \mapsto c_i]_{i=1,\ldots,n}(\alpha(\Delta!y)) & = & \widehat{\mathcal{F}}\,[\alpha(\rho_i) \mapsto c_i]_{i=1,\ldots,n}(\alpha(\rho_k)) = c_k \end{array}$$

follow, hence $\widehat{\mathcal{E}}[y_i \mapsto c_i]_{i=1,\ldots,n}(y) \sim_{\Delta!y} \widehat{\mathcal{F}}\,[\alpha(\rho_i) \mapsto c_i]_{i=1,\ldots,n}(\alpha(\Delta!y))$.

iii) For $y \in fv(\widehat{a}) \cap dom\,\Pi$, $\widehat{\mathcal{E}}[y_i \mapsto c_i]_{i=1,\ldots,n}(y) \sim_{\Pi(y)} \widehat{\mathcal{F}}\,[\alpha(\rho_i) \mapsto c_i]_{i=1,\ldots,n}(y)$ holds: from $y \in dom\,\Pi$ and $\{y_1,\ldots,y_n\} \subseteq dom\,\Delta$ we deduce that $y \notin \{y_1,\ldots,y_n\}$ holds, since the well-formedness of $\mathcal{D}^*$ implies $dom\,\Delta \cap dom\,\Pi = \emptyset$. Also, $y \neq \alpha(\rho_i)$ for all i, since α is an allocation for $\mathcal{D}^*$. Thus,

$$\widehat{\mathcal{E}}[y_i \mapsto c_i]_{i=1,\ldots,n}(y) = \widehat{\mathcal{E}}(y) \text{ and } \widehat{\mathcal{F}}\,[\alpha(\rho_i) \mapsto c_i]_{i=1,\ldots,n}(y) = \widehat{\mathcal{F}}\,(y)$$

and the claim follows by item IV) above.

Case LET. For $a \equiv \texttt{let}\ x = t\ \texttt{in}\ \widehat{a}$, the derivation for $\mathcal{E} \vdash a \Downarrow v$ has shape

$$\text{LET}\ \frac{\mathcal{E} \vdash t \Downarrow v' \quad \mathcal{E}[x \mapsto v'] \vdash \widehat{a} \Downarrow v}{\mathcal{E} \vdash a \Downarrow v}$$

for some v', derivation $\mathcal{D}$ has shape

$$\text{Let}\ \frac{\mathcal{D}_1 : \Gamma \vdash_\Sigma t : \rho \quad \mathcal{D}_2 : \Gamma, x : \rho_1 \vdash_\Sigma \widehat{a} : \rho_2 \quad x \notin dom\,\Sigma}{\Gamma \vdash_\Sigma a : \rho_2}$$

and $\mathcal{A}_\alpha(\mathcal{D}) = \texttt{let}\ \alpha(\rho_1) = \mathcal{A}_\alpha(\mathcal{D}_1)\ \texttt{in}\ \mathcal{A}_\alpha(\mathcal{D}_2)$. Applying the induction hypothesis to $\mathcal{D}_1$ (notice that $\mathcal{E} \approx^{(\Gamma,\Sigma)}_{fv(t)} \mathcal{F}$ follows from $fv(t) \subseteq fv(a)$) yields the existence of some w' with

$$\mathcal{F} \vdash \mathcal{A}_\alpha(\mathcal{D}_1) \Downarrow w' \text{ and } v' \sim_\rho w',$$

hence $w' = v' = c$ for some c by the definition of $\sim$. From $fv(\widehat{a}) \subseteq fv(a) \cup \{x\}$ and $\mathcal{E} \approx^{(\Gamma,\Sigma)}_{fv(a)} \mathcal{F}$ we now obtain

$$\mathcal{E}[x \mapsto c] \approx^{(\Gamma,x:\rho_1,\Sigma)}_{fv(\widehat{a})} \mathcal{F}\,[\alpha(\rho_1) \mapsto c]$$

as follows.

i) $fv(\widehat{a}) \subseteq (dom\ \mathcal{E}[x \mapsto c]) \cap (dom\ \Gamma, x : \rho_1 \cup dom\ \Sigma)$ holds: we have

$$\begin{array}{rcl} fv(\widehat{a}) & \subseteq & fv(a) \cup \{x\} \\ & \subseteq & (dom\ \mathcal{E} \cap (dom\ \Gamma \cup dom\ \Sigma)) \cup \{x\} \\ & \subseteq & (dom\ \mathcal{E} \cup \{x\}) \cap (dom\ \Gamma \cup dom\ \Sigma \cup \{x\}) \\ & \subseteq & (dom\ \mathcal{E}[x \mapsto c]), \end{array}$$

and $fv(\widehat{a}) \subseteq (dom\ \Gamma, x : \rho_1 \cup dom\ \Sigma)$ follows from $\mathcal{D}_2 : \Gamma, x : \rho_1 \vdash_\Sigma \widehat{a} : \rho_2$ using Lemma 1.

ii) For $z \in fv(\widehat{a}) \cap dom\ \Gamma, x : \rho_1$,

$$\mathcal{E}[x \mapsto c](z) \sim_{\Gamma, x:\rho_1 ! z} \mathcal{F}[\alpha(\rho_1) \mapsto c](\alpha(\Gamma, x : \rho_1 ! z))$$

holds: we have

- If $z = x$: $\mathcal{E}[x \mapsto c](z) = c$, $\Gamma, x : \rho_1 ! z = \rho_1$ and $\mathcal{F}[\alpha(\rho_1) \mapsto c](\alpha(\rho_1)) = c$, hence $\mathcal{E}[x \mapsto c](z) \sim_{\rho_1} \mathcal{F}[\alpha(\rho_1) \mapsto c](\alpha(\rho_1))$ holds because of $c \sim_{\rho_1} c$
- If $z \neq x$ then $\mathcal{E}[x \mapsto c](z) = \mathcal{E}(z)$ and $\Gamma, x : \rho_1 ! z = \Gamma ! z = \rho_3$ for some $\rho_3 \neq \rho_1$, hence $\mathcal{F}[\alpha(\rho_1) \mapsto c](\alpha(\rho_3)) = \mathcal{F}(\alpha(\rho_3))$ holds because $\alpha(\rho_1) \neq \alpha(\rho_3)$. Using $z \in fv(a)$ and $\mathcal{E} \approx^{(\Gamma,\Sigma)}_{fv(a)} \mathcal{F}$ we obtain $\mathcal{E}(z) \sim_{\rho_3} \mathcal{F}(\alpha(\rho_3))$, hence $\mathcal{E}[x \mapsto c](z) \sim_{\rho_3} \mathcal{F}[\alpha(\rho_1) \mapsto c](\alpha(\rho_3))$.

iii) For $z \in fv(\widehat{a}) \cap dom\ \Sigma$, $\mathcal{E}[x \mapsto c](z) \sim_{\Sigma(z)} \mathcal{F}[\alpha(\rho_1) \mapsto c](z)$ holds: we have $x \neq z$ (because of $x \notin dom\ \Sigma$ and the well-formedness of $\mathcal{D}$), hence $z \in fv(a) \cap dom\ \Sigma$ and $\mathcal{E}[x \mapsto c](z) = \mathcal{E}(z)$, and from $cod\ \alpha \cap dom\ \Sigma = \emptyset$ (as α is an allocation for $\mathcal{D}$) we obtain $\alpha(\rho_1) \neq z$, hence $\mathcal{F}[\alpha(\rho_1) \mapsto c](z) = \mathcal{F}(z)$. From $\mathcal{E} \approx^{(\Gamma,\Sigma)}_{fv(a)} \mathcal{F}$ we obtain $\mathcal{E}(z) \sim_{\Sigma(z)} \mathcal{F}(z)$, hence $\mathcal{E}[x \mapsto c](z) \sim_{\Sigma(z)} \mathcal{F}[\alpha(\rho_1) \mapsto c](z)$.

We can thus apply the induction hypothesis to $\widehat{\mathcal{E}}[x \mapsto v'] \vdash \widehat{a} \Downarrow v$, so from $\mathcal{D}_2$ we obtain the existence of some w with

$$\mathcal{F}[\alpha(\rho_1) \mapsto c] \vdash \mathcal{A}_\alpha(\mathcal{D}_2) \Downarrow w \text{ and } v \sim_{\rho_2} w.$$

The claim now follows from

$$\text{LET}\ \frac{\mathcal{F} \vdash \mathcal{A}_\alpha(\mathcal{D}_1) \Downarrow c \quad \mathcal{F}[\alpha(\rho_1) \mapsto c] \vdash \mathcal{A}_\alpha(\mathcal{D}_2) \Downarrow w}{\mathcal{F} \vdash \mathcal{A}_\alpha(\mathcal{D}) \Downarrow w}$$

Cases TRUE and FALSE. For $a \equiv \texttt{if}\ t\ \texttt{then}\ a_1\ \texttt{else}\ a_2$ the derivation of $\mathcal{E} \vdash a \Downarrow v$ has shape

$$\frac{\mathcal{E} \vdash t \Downarrow v' \quad \mathcal{E} \vdash a' \Downarrow v}{\mathcal{E} \vdash a \Downarrow v}$$

where $v' \in \{\mathbf{tt}, \mathbf{ff}\}$, $a' = a_1$ if $v' = \mathbf{tt}$ and $a' = a_2$ if $v' = \mathbf{ff}$. The derivation $\mathcal{D}$ has shape

$$\text{If}\ \frac{\mathcal{D}_0 : \Gamma \vdash_\Sigma t : \rho \quad \forall i \in \{1,2\}.\ \mathcal{D}_i : \Gamma \vdash_\Sigma a_i : \rho_1}{\Gamma \vdash_\Sigma a : \rho_1}$$

and $\mathcal{A}_\alpha(\mathcal{D}) = \texttt{if}\ \mathcal{A}_\alpha(\mathcal{D}_0)\ \texttt{then}\ \mathcal{A}_\alpha(\mathcal{D}_1)\ \texttt{else}\ \mathcal{A}_\alpha(\mathcal{D}_2)$. From $fv(t) \subseteq fv(a)$ and $\mathcal{E} \approx^{(\Gamma,\Sigma)}_{fv(a)} \mathcal{F}$ we obtain $\mathcal{E} \approx^{(\Gamma,\Sigma)}_{fv(t)} \mathcal{F}$, hence applying the induction hypothesis to $\mathcal{E} \vdash t \Downarrow v'$ yields

$$\mathcal{F} \vdash \mathcal{A}_\alpha(\mathcal{D}_0) \Downarrow w'$$

for some w' with $v' \sim_\rho w'$, so $w' = v'$ by the definition of $\sim$. From $fv(a') \subseteq fv(a)$ and $\mathcal{E} \approx^{(\Gamma,\Sigma)}_{fv(a)} \mathcal{F}$ we obtain $\mathcal{E} \approx^{(\Gamma,\Sigma)}_{fv(a')} \mathcal{F}$, hence applying the induction hypothesis to $\mathcal{E} \vdash a' \Downarrow v$ yields

$$\mathcal{F} \vdash \mathcal{A}_\alpha(\mathcal{D}_i) \Downarrow w$$

for some w with $v \sim_{\rho_1} w$, where $i = 1$ if $v' = \mathbf{tt}$ and $i = 2$ if $v = \mathbf{ff}$. The claim follows from

$$\frac{\mathcal{F} \vdash \mathcal{A}_\alpha(\mathcal{D}_0) \Downarrow v' \quad \mathcal{F} \vdash \mathcal{A}_\alpha(\mathcal{D}_i) \Downarrow w}{\mathcal{E} \vdash \mathcal{A}_\alpha(\mathcal{D}) \Downarrow w}\,.$$

- The opposite direction is similar.

A.3 Proof of Theorem 4.2

Proof Induction on the height N of $\Phi_\omega(\mathcal{D})$. We write $\mathcal{A}_\alpha(\mathcal{D}) \equiv f(\alpha(\sigma_1),\ldots,\alpha(\sigma_n))$ where $\Gamma!y_i = \sigma_i$.

Case $N = 0$**.** The last rule in $\Phi_\omega(\mathcal{D})$ was $\Phi - \mathrm{I}$, so we have $\Phi_\omega(\mathcal{D}) = \mathcal{D}$ and $\rho_j = \sigma_j$ is satisfied for all j. Therefore, $\mathcal{A}_\alpha(\Phi_\omega(\mathcal{D})) = \mathcal{A}_\alpha(\mathcal{D}) \equiv f(\alpha(\sigma_1),\ldots,\alpha(\sigma_n))$ and $\forall j.\ \alpha(\rho_j) = \alpha(\sigma_j)$ follow and the claim holds by rule G-I.

Case $N > 0$**.** There are two cases, depending on the last rule applied to obtain $\Phi_\omega(\mathcal{D})$.

Case $\Phi - \mathrm{II}$**.** We have $\mathcal{A}_\alpha(\Phi_\omega(\mathcal{D})) \equiv \mathtt{let}\ \alpha(\rho_i) = \alpha(\sigma_i)\ \mathtt{in}\ \mathcal{A}_\alpha(\Phi_\omega(\mathcal{D}^*))$, where $\rho_i \neq \sigma_j$ (hence $\alpha(\rho_i) \neq \alpha(\sigma_j)$) holds for all j, $x \notin \{y_j | j \neq i\} \cup dom\,\Sigma$, and $\mathcal{D}^*$ is the derivation

$$\frac{\forall j \neq i.\ \widehat{\mathcal{D}_j} : \Gamma, x : \rho_i \vdash_\Sigma y_j : \sigma_j \quad \Gamma, x : \rho_i \vdash_\Sigma x : \rho_i}{\Gamma, x : \rho_i \vdash_\Sigma f(y_1,\ldots,y_{i-1},x,y_{i+1},\ldots,y_n) : \rho}\,.$$

From the fact that α is an allocation for $\mathcal{D}$ we obtain that it is an allocation for $\mathcal{D}^*$, so the induction hypothesis can be applied, and we obtain

$$(\mathcal{A}_\alpha(\mathcal{D}^*), [\alpha(\rho_1),\ldots,\alpha(\rho_n)], \alpha(\omega)) \rhd \mathcal{A}_\alpha(\Phi_\omega(\mathcal{D}^*)).$$

Applying the definition of $\mathcal{A}_\alpha(.)$ yields

$$\mathcal{A}_\alpha(\mathcal{D}^*) \equiv f(\alpha(\sigma_1),\ldots,\alpha(\sigma_{i-1}),\alpha(\rho_i),\alpha(\sigma_{i+1}),\ldots,\alpha(\sigma_n)),$$

so the claim follows from the definition of $\mathcal{A}_\alpha(\mathcal{D})$ and $\mathcal{A}_\alpha(\Phi_\omega(\mathcal{D}^*))$ by rule G-II as the side condition $\alpha(\rho_i) \neq \alpha(\sigma_j)$ holds for all j.

Case $\Phi - \mathrm{III}$**.** We have $\mathcal{A}_\alpha(\Phi_\omega(\mathcal{D})) \equiv \mathtt{let}\ \alpha(\omega) = \alpha(\sigma_i)\ \mathtt{in}\ \mathcal{A}_\alpha(\Phi_\omega(\mathcal{D}^*))$, where i satisfies $\rho_i \neq \sigma_i$, and for all k there exists a j with $\rho_k = \sigma_j$. Furthermore, $x \notin \{y_j | j \neq i\} \cup dom\,\Sigma$ holds and $\mathcal{D}^*$ is the derivation

$$\frac{\forall j \text{ s.t. } y_j \neq y_i :\ \widehat{\mathcal{D}_j} : \Gamma, x : \omega \vdash_\Sigma y_j : \sigma_j \qquad \Gamma, x : \omega \vdash_\Sigma x : \omega}{\Gamma, x : \omega \vdash_\Sigma f(y_1,\ldots,y_n)[x/y_i] : \rho}\,.$$

From the identity of the sets $\{\rho_j | \rho_j \neq \sigma_j\}$ and $\{\sigma_j | \rho_j \neq \sigma_j\}$, the distinctness of the ρ_j (follows from the well-definedness of $\mathcal{D}$), the fact $\omega \notin \{\rho_1,\ldots,\rho_n\}$ and the property $\forall j.\ \Gamma!y_j = \sigma_j$, we deduce that $\omega \notin \{\sigma_1,\ldots,\sigma_n\}$ holds and that the y_j are distinct, i.e $f(y_1,\ldots,y_n)[x/y_i] \equiv f(y_1,\ldots,y_{i-1},x,y_{i+1},\ldots,y_n)$ and $\mathcal{D}^*$ actually has shape

$$\frac{\forall j \neq i :\ \widehat{\mathcal{D}_j} : \Gamma, x : \omega \vdash_\Sigma y_j : \sigma_j \qquad \Gamma, x : \omega \vdash_\Sigma x : \omega}{\Gamma, x : \omega \vdash_\Sigma f(y_1,\ldots,y_{i-1},x,y_{i+1},\ldots,y_n) : \rho}\,.$$

From the fact that α is an allocation for $\mathcal{D}$ we obtain that it is an allocation for $\mathcal{D}^*$, so the induction hypothesis can be applied to derivation $\Phi_\omega(\mathcal{D}^*)$ and we obtain

$$(\mathcal{A}_\alpha(\mathcal{D}^*), [\alpha(\rho_1), \ldots, \alpha(\rho_n)], \alpha(\omega)) \rhd \mathcal{A}_\alpha(\Phi_\omega(\mathcal{D}^*)).$$

From $\omega \notin \{\sigma_1, \ldots, \sigma_n\}$ we obtain $\alpha(\omega) \neq \alpha(\sigma_j)$ for all j, so the definition of $\mathcal{A}_\alpha(.)$ yields

$$\begin{array}{rcl} \mathcal{A}_\alpha(\mathcal{D}^*) & \equiv & f(\alpha(\sigma_1), \ldots, \alpha(\sigma_{i-1}), \alpha(\omega), \alpha(\sigma_{i+1}), \ldots, \alpha(\sigma_n)) \\ & \equiv & f(\alpha(\sigma_1), \ldots, \alpha(\sigma_n))[\alpha(\omega)/\alpha(\sigma_i)] \end{array}$$

and we obtain

$$\text{G-III}\ \frac{(\mathcal{A}_\alpha(\mathcal{D}^*), [\alpha(\rho_1), \ldots, \alpha(\rho_n)], \alpha(\omega)) \rhd \mathcal{A}_\alpha(\Phi_\omega(\mathcal{D}^*))}{(f(\alpha(\sigma_1), \ldots, \alpha(\sigma_n)), [\alpha(\rho_1), \ldots, \alpha(\rho_n)], x) \rhd \texttt{let}\ \alpha(\omega) = \alpha(\sigma_i)\ \texttt{in}\ \mathcal{A}_\alpha(\Phi_\omega(\mathcal{D}^*))}$$

as the side conditions $\alpha(\rho_i) \neq \alpha(\sigma_i)$ and $\forall k.\exists j.\alpha(\rho_k) = \alpha(\sigma_j)$ are satisfied.