Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 176 (2007) 3–20

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Functional Elimination of Φ-instructions

Lennart Beringer[1]

Lehrstuhl für Theoretische Informatik
Ludwig-Maximilians-Universität München
Oettingenstrasse 67, 80538 München, Germany

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.

Keywords: Compilation, Functional intermediate representations, Static single assignment form, Phi-elimination, Type systems for register allocation

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 [25,18,16]. 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

[1] Email: beringer@tcs.ifi.lmu.de

doi:10.1016/j.entcs.2007.06.003

$$\begin{tikzcd}
\text{SSA} \arrow[rr, leftrightarrow, "\text{Appel–Kelsey}"] \arrow[d, "\Phi\ \text{elimination}"'] & & \text{ANF} \arrow[d, "\text{GNF conversion}"] \\
\text{imp. Grail} \arrow[rr, leftrightarrow, "\text{Coincidence of semantics}"] & & \text{funct. Grail}
\end{tikzcd}$$

Fig. 1. Syntactic Grail conversion and the elimination of Φ-instructions

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].

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. 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

- 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 (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. An extended version of this paper is available from the author's home page and contains the proofs of all theorems as well as some additional material [6].

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

$$
\begin{aligned}
b &::= e \mid b; f : e \mid b; f : \{b\} \\
e &::= \texttt{ret}\ t; \mid \texttt{goto}\ f; \\
&\quad \mid x \leftarrow t; e \\
&\quad \mid x \leftarrow \Phi(p_1, \ldots, p_n); e \\
&\quad \mid \texttt{if}\ t\ \texttt{then}\ e\ \texttt{else}\ e \\
p &::= f : t \mid \texttt{start} : t
\end{aligned}
\qquad
\begin{aligned}
a \in \mathit{ANF} &::= t \mid f(t_1, \ldots, t_n) \\
&\quad \mid \texttt{let}\ x = t\ \texttt{in}\ a \\
&\quad \mid \texttt{if}\ t\ \texttt{then}\ a_1\ \texttt{else}\ a_2 \\
&\quad \mid \texttt{rec}\ \ f_1(x^1_1, \ldots, x^1_{n_1}) = a_1 \\
&\qquad\qquad \vdots \\
&\qquad\qquad f_n(x^n_1, \ldots, x^n_{n_n}) = a_n \\
&\qquad \texttt{in}\ a
\end{aligned}
$$

$$t \in \mathit{Term} ::= c \mid x$$

Fig. 2. Syntax of SSA and ANF

$f, g, \ldots, x, y \ldots$), 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 as triples of formal parameters, environment, and function body):

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

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 $\mathit{dom}\ \mathcal{E}$, 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.

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

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

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

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

Fig. 3. Operational semantics of ANF

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

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

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 (iii) is optional.

(i) all functions are fully λ-lifted.

(ii) 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.

(iii) both arms a_1 and a_2 of conditionals `if` t `then` a_1 `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. 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

(i) α-convert variables globally so that no variable is bound more than once – binding occurs in `let`-statements and function declarations.

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

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

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

Fig. 4. Rules defining $\mathcal{G}$

(ii) 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.

(iii) 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* $f(x_1,\ldots,x_n)$.

(iv) (optionally) normalise branches by inserting fresh function declarations.

The first two steps are well-known, with the two tasks in step (ii) 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

$$\texttt{let}\, y_1 = t_1\, \texttt{in}\, \ldots \texttt{let}\, y_n = t_n\, \texttt{in}$$

$$\texttt{let}\, x_1 = y_1\, \texttt{in}\, \ldots \texttt{let}\, x_n = y_n\, \texttt{in}\ f(x_1,\ldots,x_n),$$

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 $x_{i_j} = t_{i_{j+1\, mod\, l}}$ for $0 \leq j < l$).

The correctness of step (iii) 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* $\mathcal{E} \vdash f(t_1, \ldots, t_n) \Downarrow v \Longleftrightarrow \mathcal{E} \vdash a \Downarrow v$.

Proof. Induction on the height of $(f(t_1, \ldots, t_n), [x_1, \ldots, x_n], x) \rhd a$. See [6]. □

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 (iv) 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

```
let $i_1 = 1,\ j_1 = 1, k_1 = 0$ in
rec $f_2(j_2, k_2)$= if $k_2 < 100$
                     then rec $f_7(j_4, k_4) = f_2(j_4, k_4)$ in
                          if $j_2 < 20$
                          then let $j_3 = i_1,\ k_3 = k_2 + 1$ in $f_7(j_3, k_3)$
                          else let $j_5 = k_2,\ k_5 = k_2 + 1$ in $f_7(j_5, k_5)$
                     else $j_2$
in $f_2(j_1, k_1)$.
```

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

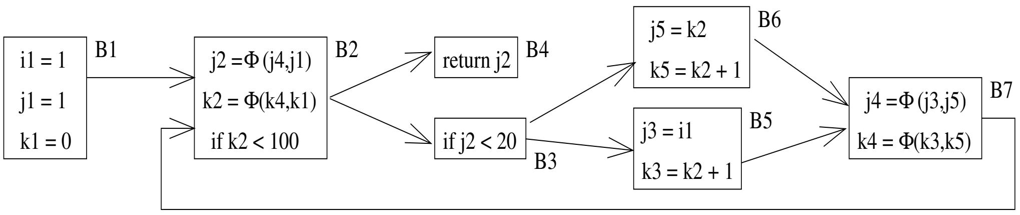

Fig. 5. Illustrating Φ-elimination (I) – program taken from [4]

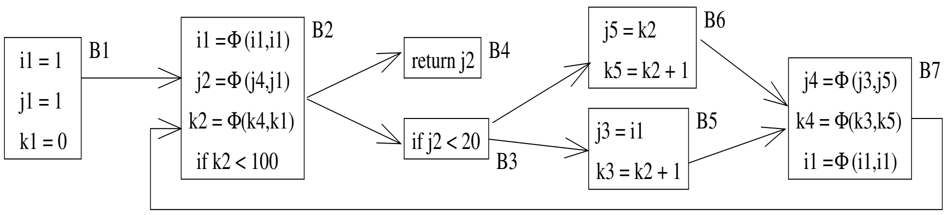

Fig. 6. Illustrating Φ-elimination (II): λ-lifting

```
rec $f_2(i_1, j_2, k_2)$ = if $k_2 < 100$
                          then if $j_2 < 20$
                               then let $j_3 = i_1,\ k_3 = k_2 + 1$ in $f_7(j_3, k_3, i_1)$
                               else let $j_5 = k_2,\ k_5 = k_2 + 1$ in $f_7(j_5, k_5, i_1)$
                          else $j_2$
    $f_7(j_4, k_4, i_1) = f_2(i_1, j_4, k_4)$
in let $i_1 = 1,\ j_1 = 1, k_1 = 0$ in $f_2(i_1, j_1, k_1)$.
```

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.

```
rec $f_2(i_1, j_2, k_2)$ =
        if $k_2 < 100$
        then if $j_2 < 20$
             then let $j_3 = i_1,\ k_3 = k_2 + 1,\ j_4 = j_3, k_4 = k_3$ in $f_7(j_4, k_4, i_1)$
             else let $j_5 = k_2,\ k_5 = k_2 + 1,\ j_4 = j_5, k_4 = k_5$ in $f_7(j_4, k_4, i_1)$
        else $j_2$
    $f_7(j_4, k_4, i_1)$ = let $j_2 = j_4, k_2 = k_4$ in $f_2(i_1, j_2, k_2)$
in let $i_1 = 1,\ j_1 = 1, k_1 = 0,\ j_2 = j_1, k_2 = k_1$ in $f_2(i_1, j_2, k_2)$
```

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.

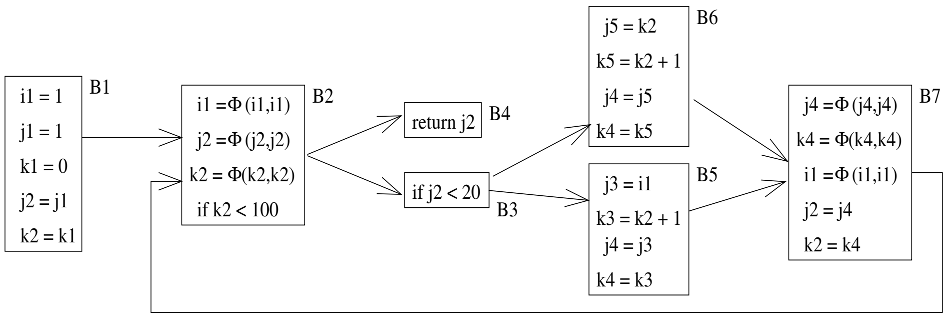

Fig. 7. Illustrating Φ-elimination (III): conversion $\mathcal{G}$

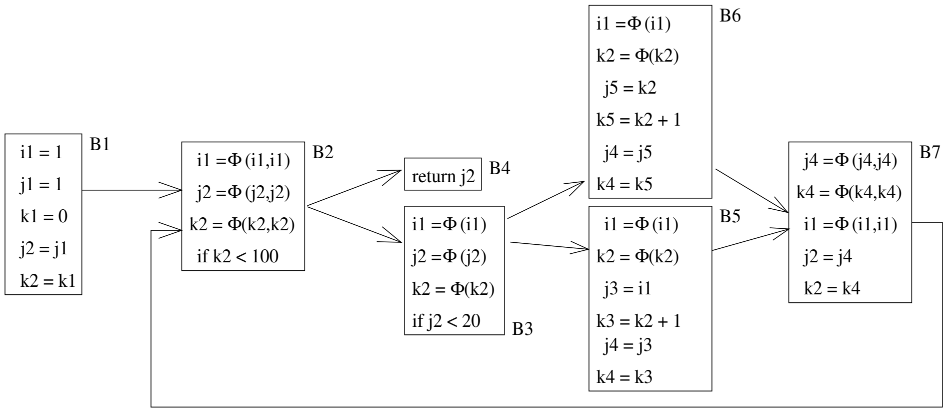

Fig. 8. Illustrating Φ-elimination (IV a): branch normalisation (part 1)

```
rec $f_2(i_1, j_2, k_2)$ = if $k_2 < 100$
                   then rec $f_3(i_1, j_2, k_2)$=
                        if $j_2 < 20$
                        then rec $f_5(i_1, k_2)$ = let $j_3 = i_1,\ k_3 = k_2 + 1,$
                                                       $j_4 = j_3,\ k_4 = k_3$
                                                   in $f_7(j_4, k_4, i_1)$
                             in $f_5(i_1, k_2)$
                        else rec $f_6(i_1, k_2)$= let $j_5 = k_2,\ k_5 = k_2 + 1,$
                                                      $j_4 = j_5, k_4 = k_5$
                                                  in $f_7(j_4, k_4, i_1)$
                             in $f_6(i_1, k_2)$
                      in $f_3(i_1\ j_2\ k_2)$
                   else $j_2$
    $f_7(j_4, k_4, i_1)$ = let $j_2 = j_4, k_2 = k_4$ in $f_2(i_1, j_2, k_2)$
in let $i_1 = 1,\ j_1 = 1, k_1 = 0,\ j_2 = j_1, k_2 = k_1$ in $f_2(i_1, j_2, k_2)$
```

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

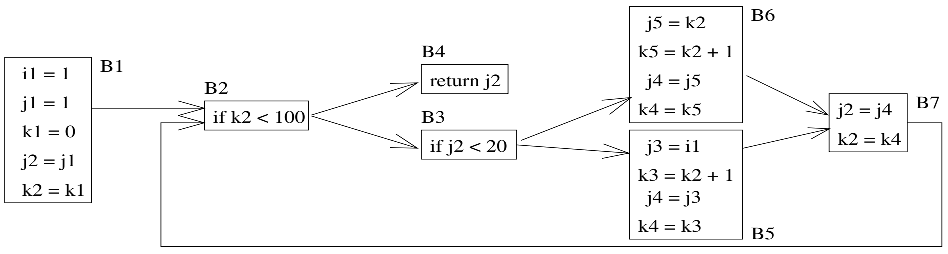

Fig. 9. Illustrating Φ-elimination (IV b): branch normalisation (part 2)

```
rec $f_2(i_1, j_2, k_2) = $ if $k_2 < 100$ then $f_3(i_1, j_2, k_2)$ else $j_2$
    $f_3(i_1, j_2, k_2) = $ if $j_2 < 20$ then $f_5(i_1, k_2)$ else $f_6(i_1, k_2)$
    $f_5(i_1, k_2) = $ let $j_3 = i_1,\ k_3 = k_2 + 1,\ j_4 = j_3, k_4 = k_3$ in $f_7(j_4, k_4, i_1)$
    $f_6(i_1, k_2) = $ let $j_5 = k_2,\ k_5 = k_2 + 1,\ j_4 = j_5, k_4 = k_5$ in $f_7(j_4, k_4, i_1)$
    $f_7(j_4, k_4, i_1) = $ let $j_2 = j_4, k_2 = k_4$ in $f_2(i_1, j_2, k_2)$
in let $i_1 = 1,\ j_1 = 1, k_1 = 0,\ j_2 = j_1, k_2 = k_1$ in $f_2(i_1, j_2, k_2)$
```

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 `let` $x = x$ `in` ... are inserted, the resulting code could be further optimised 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 [2]. 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.

In [6], we show that our algorithm respects the concurrent interpretation of Φ-instructions, using standard examples from the literature [8].

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

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

[2] 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.

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

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

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

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

Fig. 10. Rules for type system $\mathfrak{T}$

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

Fig. 11. Example typing derivation

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 $\mathit{dom}\ \Gamma = \{x | \exists \rho.\ \Gamma!x = \rho\}$, $\mathit{cod}\ \Gamma = \{\rho | \exists x.\ \Gamma!x = \rho\}$ and all operations on signatures are defined as usual. Notice that in particular, $\Gamma, x : \rho, \Delta!x = \rho$ implies $\rho \notin \mathit{cod}\ \Delta$. The typing rules for the first type system, $\mathfrak{T}$, are given in Figure 10. 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 11 shows the derivation of

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

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 4.1 A signature Σ is well-formed if for all $x \in \mathit{dom}\ \Sigma$ 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 $\mathit{dom}\ \Gamma \cap \mathit{dom}\ \Sigma = \emptyset$.

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 $\Sigma(x) = (\texttt{r1}, \texttt{r2}) \to \texttt{r2}$ is well-typed exactly if $\Gamma \vdash_\Sigma a : \texttt{r1}$ and $\Gamma \vdash_\Sigma b : \texttt{r2}$ hold, where Definition 4.1 ensures $\texttt{r1} \neq \texttt{r2}$.

Example 4.2 Consider two function definitions for the factorial function.

```
rec fac1(n, a)  =  let test  =  n < 1 in
                   if test then a else let m = n − 1,  b = a ∗ n in fac1(m, b)
rec 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` cannot be typed in system $\mathfrak{T}_\rho$, since $\Sigma(\texttt{fac1}) = (\rho_{\texttt{n}}, \rho_{\texttt{a}}) \to \rho$ yields $\rho_{\texttt{n}} \neq \rho_{\texttt{a}}$, so from rule Call we obtain $\rho_{\texttt{n}} = \rho_{\texttt{m}}$ and $\rho_{\texttt{a}} = \rho_{\texttt{b}}$, while the typing of the `else`-branch requires us to derive $\texttt{n} : \rho_{\texttt{n}}, \texttt{a} : \rho_{\texttt{a}}, \texttt{test} : \rho_{\texttt{test}}, \texttt{m} : \rho_{\texttt{m}} \vdash \texttt{n} : \rho$ for some ρ, hence $\rho_{\texttt{n}} \neq \rho_{\texttt{m}}$. In contrast, program `fac2` is typeable using the three registers $\rho_{\texttt{n}} = \rho_{\texttt{m}}$, $\rho_{\texttt{a}} = \rho_{\texttt{b}}$ and $\rho_{\texttt{test}}$.

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.

Definition 4.3 An injective map $\alpha : \mathit{Regs} \to \mathit{Var}$ is called a *register allocation* for $\mathcal{D}$ if all sequents $\Gamma \vdash_\Sigma a : \tau$ in $\mathcal{D}$ satisfy $\mathit{cod}\ \alpha \cap \mathit{dom}\ \Sigma = \emptyset$.

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 12, which converts a typing derivation $\mathcal{D}$ with final sequent $\Gamma \vdash_\Sigma a : \rho$ into $\mathcal{A}_\alpha(a)$, given allocation α.

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

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

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

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

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

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

Fig. 12. Register allocation: rewriting step

Example 4.4 For an allocation α with $\alpha(\rho_{\texttt{n}}) = \alpha(\rho_{\texttt{m}}) = \texttt{n}$, $\alpha(\rho_{\texttt{a}}) = \alpha(\rho_{\texttt{b}}) = \texttt{a}$, and $\alpha(\rho_{\texttt{test}}) = \texttt{test}$, the result of applying $\mathcal{A}_\alpha(.)$ to `fac2` is

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

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 in the following sense.

Theorem 4.5 *Let $\mathcal{D} : \Gamma \vdash_\Sigma a : \tau$ be well-formed, α a register allocation for $\mathcal{D}$, and $fv(a) = \emptyset$. Then $\mathcal{E} \vdash a \Downarrow v \iff \mathcal{F} \vdash \mathcal{A}_\alpha(\mathcal{D}) \Downarrow v$.*

The claim follows from a more general result for expressions with free variables, whose proof employs an equivalence relation on environments, following the coinductive proof technique by Milner and Tofte. For the details, see [6].

4.3 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 13, we give a definition if $\forall j.\ \rho_j = \sigma_j : \mathcal{D}$

if $\forall j.\ \rho_i \neq \sigma_j$ and $x \notin \{y_j | j \neq i\} \cup dom\ \Sigma$:

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

if $\rho_i \neq \sigma_i$ and $\forall k. \exists j. \rho_k = \sigma_j$, and $x \notin \{y_j | j \neq i\} \cup dom\ \Sigma$:

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

Fig. 13. Proof transformation Φ_ω for converting $\mathcal{D} \in \mathfrak{T}_x$ to $\Phi_\omega(\mathcal{D}) \in \mathfrak{T}_{x,\rho}$.

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 4.6 *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*

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

Proof. Induction on the height of $\Phi_\omega(\mathcal{D})$. Details are given in [6]. □

Together with Theorems 2.1 and 4.5, 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 [28,1,2,24]. 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[10,8]. 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 constancy information, information flow and dead code elimination in an imperative setting, exploiting the equational theory arising from interpretations of types as partial equivalence relations.

Acknowledgement

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, 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 (full version). Available at `http://www.tcs.ifi.lmu.de/~beringer`.

[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.