COCV'03 Preliminary Version

A Functional Perspective on SSA Optimisation Algorithms

Manuel M. T. Chakravarty [1], Gabriele Keller [1] and
Patryk Zadarnowski [1]

School of Computer Science and Engineering
University of New South Wales
Sydney, Australia

Abstract

The *static single assignment (SSA) form* is central to a range of optimisation algorithms relying on data flow information, and hence, to the correctness of compilers employing those algorithms. It is well known that the SSA form is closely related to lambda terms (i.e., functional programs), and, considering the large amount of energy expended on theories and frameworks for formal reasoning in the lambda calculus, it seems only natural to leverage this connection to improve our capabilities to reason about compiler optimisations. In this paper, we discuss a new formalisation of the mapping from SSA programs to a restricted form of lambda terms, called *administrative normal form (ANF)*. We conjecture that this connection improves our ability to reason about SSA-based optimisation algorithms and provide a first data point by presenting an ANF variant of a well known SSA-based conditional constant propagation algorithm.

---

1 Introduction

The *static single assignment (SSA) form* is a popular intermediate representation for compiler optimisations [7], and hence, of considerable significance when it comes to reasoning about the correctness of those optimisations. Unfortunately, a formal treatment of the semantics of SSA programs and SSA-based optimisation algorithms is complicated due to, the so-called ϕ-functions, which control the merging of data flow edges entering code blocks [4].

Kelsey [12] and Appel [3,2] pointed to a correspondence between programs in SSA form and lambda terms (i.e., functional programs). We believe that this correspondence can be leveraged to simplify reasoning about compiler optimisations that hitherto were based on the SSA form. In particular, we

---

[1] Email: {`chak,keller,patrykz`}`@cse.unsw.edu.au`

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

suggest that intermediate forms based on the lambda calculus lead to clearer algorithms, which we expect to positively impact the correctness of concrete implementations, even if they are not formally verified. In this paper, we concentrate on a restricted form of lambda terms, called *administrative normal form (ANF)* [8], as they have a more clearly defined operational interpretation than general lambda terms.

Kelsey [12] related the SSA form to a different form of restricted lambda terms, called *continuation passing style (CPS).* However, Flanagan et al. [8,17] showed that, for data flow analysis, there is no real advantage to using CPS over direct-style representations, such as ANF. In fact, CPS requires additional transformations and, without special measures, non-distributive flow analysis in CPS is more costly than necessary. Hence, instead of using Kelsey's CPS-based approach, we prefer to formalise the mapping of programs from SSA to ANF (in Section 2); we do so more formally than Appel [2].

Section 3 exploits the correspondence of SSA and ANF by rephrasing Wegman and Zadeck's [22] *sparse conditional constants* algorithm, which performs constant propagation and unreachable code elimination. In the following, we call Wegman and Zadeck's original SSA-based algorithm $\mathsf{Scc}_{\text{SSA}}$ and call our new ANF-based algorithm $\mathsf{Scc}_{\text{ANF}}$. We present $\mathsf{Scc}_{\text{ANF}}$ in a notation that has a well-defined semantics, as opposed to the informal notation used by Wegman and Zadeck. We believe that the semantic rigour of our notation in combination with the well-defined semantics of our intermediate language (namely ANF) implies that $\mathsf{Scc}_{\text{ANF}}$ is significantly better suited to formal analysis.

In summary, this paper makes the following technical contributions:

- We formalise the mapping of programs in SSA form to programs in ANF (Section 2.3).
- We introduce the algorithm $\mathsf{Scc}_{\text{ANF}}$, of which we claim that it implements the same analysis on ANF programs as $\mathsf{Scc}_{\text{SSA}}$ does on SSA programs (Section 3). However, $\mathsf{Scc}_{\text{ANF}}$ is more rigorously defined.
- We establish that $\mathsf{Scc}_{\text{ANF}}$ is conservative; i.e., that the variables marked as constant are indeed constant (Section 4).

However, we do not actually prove that $\mathsf{Scc}_{\text{ANF}}$ and $\mathsf{Scc}_{\text{SSA}}$ implement the same analysis. In fact, this would be hard to achieve due to Wegman and Zadeck's [22] rather informal presentation. We do, however, present formal statements about the soundness of our mapping of SSA programs to ANF programs and about the soundness of $\mathsf{Scc}_{\text{ANF}}$.

We like to regard the results in this paper as a step towards reaping the well established benefits of typed, functional intermediate languages in compilers for conventional languages. These benefits include simplified reasoning about the correctness of compiler optimisations, type-based validation of optimised code at compile time, and support for the generation of certified binaries [9,20,16,14,18]. Moreover, ANF naturally integrates intra-procedural with inter-procedural analysis.

2 Making Dataflow Explicit

In this section, we define a concrete notation for both SSA and ANF. In addition, we characterise the relationship between these two intermediate forms by a translation procedure that takes programs in SSA form into equivalent programs in ANF.

2.1 Static Single Assignment Form

The static single assignment (SSA) form [1] is an imperative representation of programs which encodes data flow information explicitly by requiring that there be exactly one assignment for every variable. Figure 2 presents the factorial function in SSA form. Except for the two ϕ-functions, the program resembles standard three-address code: "fac" consists of two basic blocks connected by jumps, with the second (labelled L_1) constituting the main loop of the program. For brevity, we allow inlining of blocks in the branches of a conditional statement; in reality, the two assignments in the `if` statement should be placed in a separate block, so that `if` indeed represents a conditional jump.

The values of the variables x and r are updated in three places in "fac". To achieve the single assignment property, we create a new variable for each update, splitting x into x, x_0 and x_1, and similarly for r. Since, the block L_1 can be reached either from the start block or from L_1 itself, at the beginning of L_1 we must merge the two sources of values for x and r using the so-called ϕ-function, which selects a value based on the source block of the jump. Note that, in SSA, the single-assignment property is purely syntactic, since, in the loop L_1, variables are still updated at runtime on every iteration.

A procedure can be put in the SSA form by arranging the basic blocks into a *dominator tree*, where the parent of each node dominates its children. An assignment *dominates* an expression if every path to the expression from the start of the procedure includes that assignment. The SSA form has been popularised by Cytron et al. [7], who describe an algorithm for computing the dominance information in linear time and demonstrate how SSA increases the power of many optimisations which require data-flow analysis.

Figure 1 presents an abstract syntax of the *structured SSA form*; i.e., to our variant of SSA in which the dominator tree is explicitely encoded in the block structure. The body of each procedure consists of a sequence of labelled expressions structured into the dominator tree (the first block in each braced group dominates the remaining blocks in that group). Intuitively, the braces provide a traditional scoping hierarchy for block labels. Further, instead of packing variables into a CFG, we follow Kelsey [12] in annotating each parameter to a ϕ function with the label of the basic block that computes the corresponding value, or `start` when the value is computed in the unlabelled entry block of the procedure.

A complete SSA program p consists of a set of (possibly-recursive) procedures and an entry point e. An SSA expression consists of a sequence of

$$
\begin{array}{rcl}
p & ::= & \texttt{proc}\ x\,\texttt{(}\overline{x}\texttt{)}\ \texttt{\{}b\texttt{\}}\ p \mid e \\
b & ::= & e \mid b\texttt{;}\ x\texttt{:}e \mid b_1\texttt{;}\ x\texttt{:\{}b_2\texttt{\}} \\
e & ::= & x \leftarrow \phi(\overline{g})\ \texttt{;}\ e \mid \\
  &   & x \leftarrow v\texttt{;}\ e \mid x \leftarrow v\,\texttt{(}\overline{v}\texttt{)}\ \texttt{;}\ e \mid \\
  &   & \texttt{goto}\ x\texttt{;} \mid \\
  &   & \texttt{ret}\ v\texttt{;} \mid \texttt{ret}\ v\,\texttt{(}\overline{v}\texttt{)}\ \texttt{;} \mid \\
  &   & \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\
g & ::= & l\texttt{:}v \\
l & ::= & x \mid \texttt{start} \\
v & ::= & x \mid c \\
\overline{x} & ::= & x\,\texttt{,}\ \overline{x} \mid \epsilon \\
\overline{v} & ::= & v\,\texttt{,}\ \overline{v} \mid \epsilon \\
\overline{g} & ::= & g\,\texttt{,}\ \overline{g} \mid \epsilon \\
x & ::= & \text{variable or label} \\
c & ::= & \text{constant}
\end{array}
$$

(SSA)

$$
\begin{array}{rcl}
e & ::= & v \mid v\,\texttt{(}\overline{v}\texttt{)} \mid \\
  &   & \texttt{let}\ x\ \texttt{=}\ v\ \texttt{in}\ e \mid \\
  &   & \texttt{let}\ x\ \texttt{=}\ v(\overline{v})\ \texttt{in}\ e \mid \\
  &   & \texttt{letrec}\ \overline{f}\ \texttt{in}\ e_S \mid \\
  &   & \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\
f & ::= & x(\overline{x})\ \texttt{=}\ e \\
v & ::= & x \mid c \\
\overline{x} & ::= & x\,\texttt{,}\ \overline{x} \mid \epsilon \\
\overline{v} & ::= & v\,\texttt{,}\ \overline{v} \mid \epsilon \\
\overline{f} & ::= & f\texttt{;}\ \overline{f} \mid \epsilon \\
x & ::= & \text{variable} \\
c & ::= & \text{constant}
\end{array}
$$

(ANF)

Fig. 1. Static Single Assignment and A-normal Forms

assignments ending with a jump. For the purpose of this discussion, we leave the set of constants unspecified; in general, it will include integers, floating point numbers and machine opcodes (primitives). For simplicity, we assume that all variables and labels in a program are unique.

2.2 Administrative Normal Form

The right-hand side of Figure 1 presents an abstract syntax of programs in the *administrative normal form* (ANF) [8], a direct-style form of lambda terms which, like SSA, restricts function parameters to atomic expressions. As demonstrated by the example ANF code in Figure 2, functions are introduced using `letrec` expressions, which correspond both to a complete program and the CFG structure of a particular procedure in the SSA form. Tail calls are made explicit by their placement within a `let` expression: function applications appearing on the right-hand side of a variable binding represent normal calls, while those appearing in the body represent tail calls (jumps).

Like the SSA form, ANF encodes data flow explicitly by naming all subexpressions within the program and permitting only a single definition of any particular variable. However, in ANF this restriction is dynamic, since, at runtime, a new scope is created for every invocation of a function. This makes formulation of the ANF semantics straight-forward and intuitive. Further, it reduces the number of mechanisms present for expressing data flow from two (ϕ-functions and procedures parameters) to one (parameters only), simplifying program analysis and eliminating the artificial distinction between intra- and inter-procedural data flow in SSA. Finally, the syntactically-clear scoping of definitions simplifies formulation of many valid and useful optimisations that involve code motion across basic blocks. In SSA, design of those algorithms is hampered by the need to preserve the dominance property of a program [2]. A formal operational semantics of SSA and ANF programs are presented in the

```
proc fac(x) {
      r ← 1;
      goto L1;;
  L1: r0 ← φ(start:r, L1:r1);
      x0 ← φ(start:x, L1:x1);
      if x0 then
            r1 ← mul(r0, x0);
            x1 ← sub(x0, 1);
            goto L1;
      else
            ret r0;}
ret fac(10);
```

(SSA)

```
letrec
  fac(x) =
    letrec
      fac′(x0,r0) =
        if x0 then
          let r1 = mul(r0, x0)
              x1 = sub(x0, 1)
          in fac′(x1, r1)
        else
          r0
      in fac′(x, 1)
 in fac(10)
```

(ANF)

Fig. 2. SSA and ANF representations of the factorial function

unabridged version of this paper [5]. Variants of ANF are used as the intermediate representation in many compilers for functional languages, including GHC [11,21] for Haskell and TIL [20] for ML.

2.3 From SSA to ANF

The similarities between the SSA and ANF forms of the factorial function in Figure 2 are immediately obvious: SSA blocks are translated into ANF functions, with the list of arguments derived from the list of ϕ-expressions in the block. In Figure 3, we define a function $\mathcal{F}$ which formalises the translation of a well-formed structured SSA program into ANF.

The translation follows the structure of a program in the SSA form. The complete program is translated by $\mathcal{F}$ and $\mathcal{F}_p$ into an all-encompassing outermost `letrec`. Within each procedure, $\mathcal{F}_b$ and $\mathcal{F}_l$ generate an ANF function for every SSA block, with each level of the dominator tree translated into a separate nested `letrec`. This makes variables defined along the dominator path visible through the usual scoping rules for nested procedures, while constructing a new dynamic scope for each iteration through a translated SSA block. The dominator of a group is selected for the body expression of the `letrec`. Further, since the leaves of the dominator tree cannot be reached except through their immediate dominator, the scoping rules for ANF enforce the dominance property of the SSA form. If desired, the nested `letrec` structure in the resulting program can be flattened using the standard lambda-lifting [10] transformation.

Each SSA block is translated into a separate ANF function. A jump is translated into a tail call to the corresponding function, with the list of parameters obtained by $\mathcal{F}_g$ from the list of ϕ-nodes in the destination block. When translating a block into a function, the corresponding list of formal parameters is computed using $\mathcal{F}_\phi$. Since, in SSA, a variable not defined along the dominator path must be accessed through a ϕ-node, this enforces the well-formedness of the resulting program.

The translation function:

$\mathcal{F}(\llbracket p \rrbracket) = \texttt{letrec}\ \mathcal{F}_p(p)\ \texttt{in}\ \mathcal{F}_e(\mathcal{S}_p(p), \texttt{start}, \langle\rangle)$

Collect SSA procedures for the outer `letrec`:

$\mathcal{F}_p(\llbracket e \rrbracket) = \epsilon$

$\mathcal{F}_p(\llbracket \texttt{proc}\ x\texttt{(}\overline{x}\texttt{)}\ \texttt{\{}b\texttt{\}}\ p \rrbracket) = x\texttt{(}\overline{x}\texttt{)=}\mathcal{F}_b(b, \texttt{start}, \mathcal{B}(b, \langle\rangle))\texttt{;}\mathcal{F}_p(p)$

Construct the inner `letrec` structure from the dominator tree:

$\mathcal{F}_b(\llbracket e \rrbracket, l, B) = \texttt{letrec}\ \mathcal{F}_l(b, B)\ \texttt{in}\ \mathcal{F}_e(\mathcal{S}_b(b), l, B)$

Collect the SSA blocks into an inner `letrec` list:

$\mathcal{F}_l(\llbracket e \rrbracket, B) = \epsilon$

$\mathcal{F}_l(\llbracket b\texttt{;}\ x\texttt{:}e \rrbracket, B) = x\texttt{(}\mathcal{F}_\phi(e, \epsilon)\texttt{)=}\mathcal{F}_e(e, x, B)\texttt{;}\mathcal{F}_l(b, B)$

$\mathcal{F}_l(\llbracket b_1\texttt{;}\ x\texttt{:\{}b_2\texttt{\}} \rrbracket, B) = x\texttt{(}\mathcal{F}_\phi(\mathcal{S}_b(b_2), \epsilon)\texttt{)=}\mathcal{F}_b(b_2, x, \mathcal{B}(b_2, B))\texttt{;}\mathcal{F}_l(b_1, B)$

Convert an SSA block to an ANF expression:

$\mathcal{F}_e(\llbracket x \leftarrow \phi\texttt{(}\overline{g}\texttt{)}\texttt{;}\ e \rrbracket, l, B) = \mathcal{F}_e(e, l, B)$

$\mathcal{F}_e(\llbracket x \leftarrow v\texttt{;}\ e \rrbracket, l, B) = \texttt{let}\ x = v\ \texttt{in}\ \mathcal{F}_e(e, l, B)$

$\mathcal{F}_e(\llbracket x \leftarrow v\texttt{(}\overline{v}\texttt{)}\texttt{;}\ e \rrbracket, l, B) = \texttt{let}\ x = v\texttt{(}\overline{v}\texttt{)}\ \texttt{in}\ \mathcal{F}_e(e, l, B)$

$\mathcal{F}_e(\llbracket \texttt{ret}\ v\texttt{;} \rrbracket, l, B) = v$

$\mathcal{F}_e(\llbracket \texttt{ret}\ v\texttt{(}\overline{v}\texttt{)}\texttt{;} \rrbracket, l, B) = v\texttt{(}\overline{v}\texttt{)}$

$\mathcal{F}_e(\llbracket \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rrbracket, l, B) = \texttt{if}\ v\ \texttt{then}\ \mathcal{F}_e(e_1, l, B)\ \texttt{else}\ \mathcal{F}_e(e_2, l, B)$

$\mathcal{F}_e(\llbracket \texttt{goto}\ x\texttt{;} \rrbracket, l, B) = x\texttt{(}\mathcal{F}_g(\beta(B, x), l, \epsilon)\texttt{)}$

Construct a list of function parameters from the ϕ-nodes:

$\mathcal{F}_\phi(\llbracket x \leftarrow \phi\texttt{(}\overline{g}\texttt{)}\texttt{;}\ e \rrbracket, \overline{x}) = x\texttt{,}\mathcal{F}_\phi(e, \overline{x})$

$\mathcal{F}_\phi(\llbracket x \leftarrow v\texttt{;}\ e \rrbracket, \overline{x}) = \mathcal{F}_\phi(e, \overline{x})$

$\mathcal{F}_\phi(\llbracket x \leftarrow v\texttt{(}\overline{v}\texttt{)}\texttt{;}\ e \rrbracket, \overline{x}) = \mathcal{F}_\phi(e, \overline{x})$

$\mathcal{F}_\phi(\llbracket \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rrbracket, \overline{x}) = \mathcal{F}_\phi(e_1, \mathcal{F}_\phi(e_2, \overline{x}))$

$\mathcal{F}_\phi(\llbracket e \rrbracket, \overline{x}) = \overline{x}$

Construct an argument list for a translated jump:

$\mathcal{F}_g(\llbracket x \leftarrow \phi\texttt{(}\overline{g}\texttt{)}\texttt{;}\ e \rrbracket, l, \overline{x}) = \kappa(\overline{g}, l)\texttt{,}\mathcal{F}_g(e, l, \overline{x})$

$\mathcal{F}_g(\llbracket x \leftarrow v\texttt{;}\ e \rrbracket, l, \overline{x}) = \mathcal{F}_g(e, l, \overline{x})$

$\mathcal{F}_g(\llbracket x \leftarrow v\texttt{(}\overline{v}\texttt{)}\texttt{;}\ e \rrbracket, l, \overline{x}) = \mathcal{F}_g(e, l, \overline{x})$

$\mathcal{F}_g(\llbracket \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rrbracket, l, \overline{x}) = \mathcal{F}_g(e_1, l, \mathcal{F}_g(e_2, l, \overline{x}))$

$\mathcal{F}_g(\llbracket e \rrbracket, l, \overline{x}) = \overline{x}$

Find the entry expression for the program:

$\mathcal{S}_p(\llbracket e \rrbracket) = e$

$\mathcal{S}_p(\llbracket \texttt{proc}\ x\texttt{(}\overline{x}\texttt{)}\ \texttt{\{}b\texttt{\}}\ p \rrbracket) = \mathcal{S}_p(p)$

Find the entry block for a procedure:

$\mathcal{S}_b(\llbracket e \rrbracket) = e$

$\mathcal{S}_b(\llbracket b\ x\texttt{:}e \rrbracket) = \mathcal{S}_b(b)$

$\mathcal{S}_b(\llbracket b_1\texttt{;}\ x\texttt{:\{}b_2\texttt{\}} \rrbracket) = \mathcal{S}_b(b_1)$

Construct the label list for a procedure:

$\mathcal{B}(\llbracket e \rrbracket, B) = B$

$\mathcal{B}(\llbracket b\texttt{;}\ x\texttt{:}e \rrbracket, B) = \mathcal{B}(b, B)\texttt{,}\ x \mapsto e$

$\mathcal{B}(\llbracket b_1\texttt{;}\ x\texttt{:\{}b_2\texttt{\}} \rrbracket, B) = \mathcal{B}(b_1, B)\texttt{,}\ x \mapsto \mathcal{S}_b(b_2)$

Search the list of ϕ parameters for a label:

$\kappa(\llbracket x_1\texttt{:}v\texttt{,}\overline{g} \rrbracket, x_2) = \textbf{if}\ x_1 = x_2\ \textbf{then}\ v\ \textbf{else}\ \kappa(\overline{g}, x_2)$

Look up a label environment:

$\beta(\llbracket B\texttt{,}\ x_1 \mapsto e \rrbracket, x_2) = \textbf{if}\ x_1 = x_2\ \textbf{then}\ e\ \textbf{else}\ \beta(B, x_2)$

Auxiliary syntax for the label environment:

$B\ ::= \langle\rangle \mid B\texttt{,}\ x \mapsto e$

Fig. 3. Translation of SSA to ANF

Theorem 2.1 *For any well-formed SSA program p, $\mathcal{F}(p)$ generates a well-formed ANF program.*

The proof of Theorem 2.1 relies on well-formedness properies for SSA and ANF, which is presented in the unabridged version of this paper [5].

3 Sparse Conditional Constants in Functional Form

To demonstrate the advantages of ANF over SSA, we introduce an ANF variant of the SSA-based *sparse conditional constants* algorithm $\mathsf{Scc}_{\text{SSA}}$. Due to space constraints, we are not able to present the original algorithm in this paper; please refer to the article of Wegman & Zadeck [22] for a comparison. The reminder of this section defines $\mathsf{Scc}_{\text{ANF}}$, which we claim discovers the same constants as $\mathsf{Scc}_{\text{SSA}}$ and removes the same unreachable code, given the ANF term computed from the SSA program using the function $\mathcal{F}$ from Figure 3.

3.1 Prerequisites

We assume that the analysed program is in ANF (as defined in Figure 1), that it is first-order, and that all variable names are unique and contained in the set Var. (We discuss higher-order programs in Section 3.5.) Moreover, the set $\mathsf{Prim} \subset \mathsf{Var}$ contains the names of all primitive functions. The analysis proceeds over an abstract domain $\mathsf{Abs} = \{\bot, \top\} \cup \mathsf{Const}$, where the $c \in \mathsf{Const}$ are the constant values of the concrete value domain. A partial order $\sqsubset$ is defined on Abs, which has $\bot$ as its least element and $\top$ as its largest element.[2] More precisely, for each constant $c \in \mathsf{Const}$, we have $\bot \sqsubset c \sqsubset \top$. In addition, we define, for $x, y \in \mathsf{Abs}$, $z = x \sqcap y$ to be the least value in Abs such that $x \sqsubseteq z$ and $y \sqsubseteq z$. The intuition underlying this abstract domain is that whenever a variable or function maps to $\bot$, it never receives a concrete value;[3] whenever it maps to a constant, this is the only value it ever assumes; and whenever it maps to $\top$, it is non-constant.

We require a function $\mathcal{E}$ that implements the evaluation of primitives from Prim; i.e., for $p \in \mathsf{Prim}$ and $c_i \in \mathsf{Const}$, $\mathcal{E}[\![p(c_1, \ldots, c_n)]\!]$ computes the result of applying p to the c_i. Moreover, $\mathcal{E}_{\mathsf{Abs}}$ extends $\mathcal{E}$ to the abstract domain Abs—i.e., if any argument to the primitive is $\bot$ or $\top$, it yields $\bot$ or $\top$, respectively; otherwise, it behaves as $\mathcal{E}$.

Apart from the input program, the central data structure of the algorithm is an environment $\Gamma : \mathsf{Var} \to \mathsf{Abs}$ that maps variable names to values of the abstract domain. The environment includes entries for value and function variables, where the latter determine the result value of the function. We write

---

[2] We use the abstract interpretation convention that $\bot$ represents no result and $\top$ represents conflicting values. This is opposite to the use of the same symbols in the dataflow analysis literature.

[3] Note that this includes non-terminating functions; hence, it is not a sufficient condition for concluding that the function is dead code.

Γx to denote the lookup of the value associated with x in the environment Γ and $\Gamma[x \mapsto a]$ to denote updating of the value associated with x to be a. In addition, $\mathrm{Dom}\,\Gamma$ denotes the variables that have an entry in Γ. Finally, we define the *refinement* of the entry for x by a as

$$\begin{array}{lll} \Gamma \sqcap [x \mapsto a] & \mid x \notin \mathrm{Dom}\,\Gamma = & \Gamma[x \mapsto a] \\ & \mid \text{otherwise} \quad = & \Gamma[x \mapsto (\Gamma\, x \sqcap a)] \end{array}$$

The initial value for the environment is $\Gamma_{\mathsf{Prim}} = [p \mapsto \top \mid \forall p \in \mathsf{Prim}]$.

We use $\mathrm{FV}\, f$ to denote all variables that are free in the body of function f; note that this includes all argument variables and, in the case of a recursive function, the function name itself. Conversely, $\mathrm{Occ}\, x$ denotes all functions f, in the program, for which $x \in \mathrm{FV}\, f$. The algorithm maintains a work list Ω of names of functions that need to be processed; Ω is initially empty.

3.2 The Algorithm

The algorithm computes the optimised form of the program in two phases: The first phase, $\mathcal{A}$, analyses the program by computing the variable environment Γ and the second phase, $\mathcal{S}$, computes the optimised version based on Γ. Both phases are syntax-directed (i.e., proceed according to the ANF grammar from Fig. 1). Note that in the definition of $\mathcal{A}$ and $\mathcal{S}$, the meta-variable v may either be a constant c or a value variable x. We assume the canonical extension of Γ to constant values, so that $\Gamma c = c$.

We denote $\mathcal{A}$ and $\mathcal{S}$ using a notation that can be understood as a LaTeX-enhanced version of the purely functional programming language Haskell [15]. Consequently, the semantics of our algorithm is accurately defined by the language definition of Haskell.

3.2.1 The first phase: program analysis

The function $\mathcal{A}$, which is being displayed in Figure 4 (see the next page), gets three arguments: (1) an ANF expression e that is being traversed recursively, (2) the current variable environment Γ, and (3) the current work list Ω. It returns a triple containing (a) the abstract value of e, (b) an updated variable environment, and (c) a new work list. The latter contains functions that are used, but not defined in e, and whose usage has increased the knowledge about the range of argument values with which the functions are invoked. The computationally most costly case of $\mathcal{A}$ is that of `letrec` expressions, where we need to iterate until Γ does not collect any new information.

Given a program e in ANF, if $\mathcal{A}[\![e]\!]\ \Gamma_{\mathsf{Prim}}\ \{\} = \langle a, \Gamma, \{\}\rangle$, we can distinguish three cases:

- If $a = \bot$, the program does not terminate.
- If $a = c$, for a constant c, the program invariably results in c.
- If $a = \top$, the environment Γ characterises the usage of all variables in the

```
$\mathcal{A}[\![v]\!]\ \Gamma\,\Omega = \langle \Gamma\,v,\ \Gamma,\ \Omega\rangle$
$\mathcal{A}[\![f\,(v_1,\ldots,v_n)]\!]\ \Gamma\,\Omega$
  $\mid f \in \mathsf{Prim}$   $= \mathcal{E}_{\mathsf{Abs}}[\![f\ (\Gamma\,v_1,\ \ldots,\ \Gamma\,v_n)]\!]$
  $\mid$ otherwise   $= \langle \Gamma'\,f,\ \Gamma',\ \textbf{if}\ \mathit{changed}\ \textbf{then}\ \Omega \cup \{f\}\ \textbf{else}\ \Omega\rangle$
 where
   $f$ is defined as $f\,(x_1,\ldots,x_n)\ =\ e$
   $\Gamma' = \Gamma \sqcap [f \mapsto \bot,\ x_1 \mapsto \Gamma\,v_1,\ \ldots,\ x_n \mapsto \Gamma\,v_n]$
   $\mathit{changed} = \exists\, i.\, \Gamma\,x_i \sqsubset \Gamma\,v_i$   -- indicates whether $\Gamma$ changed
$\mathcal{A}[\![\mathtt{let}\ x\mathtt{=}e_1\ \mathtt{in}\ e_2]\!]\ \Gamma\,\Omega =$ let
      $\langle a,\ \Gamma',\ \Omega'\rangle = \mathcal{A}[\![e_1]\!]\,\Gamma\,\Omega$
      $\Gamma'' = \Gamma' \sqcap [x \mapsto a]$
      $\mathit{changed} = \Gamma\,x \sqsubset \Gamma\,a$
      $\mathit{affected} = \mathrm{Occ}\ x \cap \mathrm{Dom}\,\Gamma$
        -- $\cap\,\mathrm{Dom}\,\Gamma$ removes not yet used functions
    in
    $\mathcal{A}[\![e_2]\!]\,\Gamma''\ (\textbf{if}\ \mathit{changed}\ \textbf{then}\ \Omega \cup \mathit{affected}\ \textbf{else}\ \Omega)$
$\mathcal{A}[\![\mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2]\!]\ \Gamma\,\Omega$
  $\mid \Gamma\,v == \bot$   $= \langle \bot,\ \Gamma,\ \Omega\rangle$
  $\mid \Gamma\,v == \mathrm{True}$   $= \mathcal{A}[\![e_1]\!]\,\Gamma\,\Omega$
  $\mid \Gamma\,v == \mathrm{False}$   $= \mathcal{A}[\![e_2]\!]\,\Gamma\,\Omega$
  $\mid \Gamma\,v == \top$   $=$ let
      $\langle a_1,\ \Gamma_1,\ \Omega_1\rangle = \mathcal{A}[\![e_1]\!]\,\Gamma\,\Omega$
      $\langle a_2,\ \Gamma_2,\ \Omega_2\rangle = \mathcal{A}[\![e_2]\!]\,\Gamma_1\,\Omega_1$
    in
    $\langle a_1 \sqcap a_2,\ \Gamma_2,\ \Omega_2\rangle$
$\mathcal{A}[\![\mathtt{letrec}\ f_1,\ldots,f_n\ \mathtt{in}\ e]\!]\Gamma\,\Omega =$
  let
    $\langle a,\ \Gamma',\ \Omega'\rangle = \mathcal{A}[\![e]\!]\,\Gamma\,\{\}$
    $\langle \Gamma'',\ \Omega''\rangle = \mathcal{A}_{\mathrm{fix}}[\![f_1,\ \ldots,\ f_n]\!]\,\Gamma'\,(\Omega \cup \Omega')$
  in
  if $\Gamma' == \Gamma''$ then $\langle a,\ \Gamma',\ \Omega''\rangle$ else $\mathcal{A}[\![\mathtt{letrec}\ f_1,\ \ldots,\ f_n\ \mathtt{in}\ e]\!]\,\Gamma''\,\Omega''$

$\mathcal{A}_{\mathrm{fix}}[\![\mathit{fun}_1,\ldots,\mathit{fun}_n]\!]\,\Gamma\,\Omega \mid \nexists\, i.f_i \in \Omega = \langle \Gamma,\ \Omega\rangle$
                                   $\mid$ otherwise $=$
  let
    $\langle a,\ \Gamma',\ \Omega'\rangle = \mathcal{A}[\![e]\!]\,\Gamma\,\{\}$
    $\Gamma'' = \Gamma' \sqcap [f_i \mapsto a]$
    $\Omega'' = \Omega \cup \Omega' \setminus \{f_i\}$
  in
  $\mathcal{A}_{\mathrm{fix}}[\![\mathit{fun}_1,\ \ldots,\ \mathit{fun}_n]\!]\,\Gamma''\ (\textbf{if}\ \Gamma\,f_i \sqsubset \Gamma\,a\ \ \textbf{then}\ \Omega'' \cup (\mathrm{Occ}\ f_i \cap \mathrm{Dom}\,\Gamma)\ \textbf{else}\ \Omega'')$
 where
   $(f_i\ (x_1,\ \ldots,\ x_m)\ =\ e)\ =\ \mathit{fun}_i$
```

Fig. 4. The analysis function of $\mathsf{Scc}_{\mathrm{ANF}}$

program. In particular, all variables mapping to constant values may be replaced by said constants. Moreover, all functions $f \notin \mathrm{Dom}\,\Gamma$ constitute unreachable code.

The various equations of $\mathcal{A}$ operate as follows. If the analysed expression is simply a constant or variable, we use Γ to determine its abstract value. In the case of the application of a primitive $p(v_1, \ldots, v_n)$, we use Γ to obtain the abstract values of the arguments v_i and apply the abstract evaluation function $\mathcal{E}_{\mathsf{Abs}}$. More interesting is the case of the application of a user-defined function, $f(v_1, \ldots, v_n)$. The environment Γ is refined to Γ' by refining the mapping of each x_i (the formal parameters to f) by $\Gamma\ v_i$ (the concrete values at which f is called). If Γ actually changes during this refinement, f is added to the work list Ω, as we need to (re)process its definition in view of the new environment. The refinement $f \mapsto \bot$ is important to ensure that f occurs in Γ; i.e., it is flagged as reachable code.

If the analysed expression has the form `let` x=e_1 `in` e_2, we refine the mapping of x in Γ based on the abstract value of e_1; if this refinement changes Γ, all functions that contain x as a free variable are added to the work list, as their abstract value may change as a consequence. Note that we need to intersect $\mathrm{Occ}\ x$ with $\mathrm{Dom}\,\Gamma$ to ensure that only functions that are guaranteed to be reachable are added to the work list. If the analysed expression is a conditional `if` v `then` e_1 `else` e_2, we form the meet $a_1 \sqcap a_2$ of the abstract values of the two branches if the condition variable v is non-constant; otherwise, we choose the branch determined by v.

Finally, in the case of an expression `letrec` $f_1, \ldots, f_n$ `in` e, we first traverse the body expression e, to collect all uses of functions from the mutually recursive set of bindings $f_1, \ldots, f_n$ in the modified work list Ω'; afterwards, we analyse the f_i using the auxiliary function $\mathcal{A}_{\mathrm{fix}}$. The latter is a recursive function that, on each call, picks a function f_i that occurs in the current work list and analyses its right hand side. It uses the result to refine the entry of f_i in Γ; as in the case of plain `let` bindings, the work list is extended by $\mathrm{Occ}\ f_i$ if Γ changes. The function $\mathcal{A}_{\mathrm{fix}}$ terminates if none of the f_i occur in the work list Ω anymore. This does not mean that Ω is necessarily empty; it may still contain functions defined in enclosing `letrec` expressions.

3.2.2 The second phase: program specialisation

On the basis of the assignment of abstract values to variables in Γ, the function $\mathcal{S}$ transforms the original program into an optimised program that has constant variables and unreachable code removed. The function $\mathcal{S}$ operates in a single sweep over the program and exploits the following properties: Whenever Γ maps a variable (function) to a constant c, this variable (function) will contain (return) c in any possible run of the program that uses the variable (invokes the function). Moreover, a value of $\bot$ for a function indicates that it does not terminate. Hence, any occurrence of such a function may be replaced by an arbitrary diverging computation, which we again denote by $\bot$.

Any function not in $\mathrm{Dom}\,\Gamma$ is dead code. The specialisation function that takes the original program in combination with the environment Γ computed by $\mathcal{A}$ to the optimised program is defined as follows:

$$
\begin{array}{llcll}
\mathcal{S}[\![v]\!] & \mid \Gamma\, v \notin \{\top,\bot\} & = & \Gamma\, v & \text{-- constant}\\
 & \mid \text{otherwise} & = & v & \text{-- non-constant}\\
\mathcal{S}[\![f\,\texttt{(}v_1\texttt{,}\,\ldots\,\texttt{,}v_n\texttt{)}]\!] & \mid \Gamma\, f \neq \top & = & \Gamma\, f & \\
 & \mid \mathit{eval} & = & \mathcal{E}[\![f\ \texttt{(}w_1\texttt{,}\ \ldots\texttt{,}\ w_n\texttt{)}]\!] & \\
 & \mid \text{otherwise} & = & f\ \texttt{(}w_1\texttt{,}\ \ldots\texttt{,}\ w_n\texttt{)} & \\
\end{array}
$$

$$
\begin{array}{l}
\textbf{where}\\
\quad w_i \ = \mathcal{S}[\![v_i]\!],\ \forall\, i \in \{1,\ldots,n\}\\
\quad \mathit{eval} = f \in \mathsf{Prim}\ \textbf{and}\ \forall\, i \in \{1,\ldots,n\}.\ w_i \in \mathsf{Const}
\end{array}
$$

$$
\begin{array}{llcll}
\mathcal{S}[\![\texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ e_2]\!] & \mid \Gamma\, x \notin \{\top,\bot\} & = & \mathcal{S}[\![e_2]\!] & \text{-- } x \text{ is constant}\\
 & \mid \text{otherwise} & = & \texttt{let}\ x\texttt{=}\mathcal{S}[\![e_1]\!]\ \texttt{in}\ \mathcal{S}[\![e_2]\!] & \\
\mathcal{S}[\![\texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!] & & & & \\
\quad \mid \Gamma\, v == \text{True} & & = & \mathcal{S}[\![e_1]\!] & \\
\quad \mid \Gamma\, v == \text{False} & & = & \mathcal{S}[\![e_2]\!] & \\
\quad \mid \text{otherwise} & & = & \texttt{if}\ v\ \texttt{then}\ \mathcal{S}[\![e_1]\!]\ \texttt{else}\ \mathcal{S}[\![e_2]\!] & \\
\end{array}
$$

$$
\begin{array}{llcll}
\mathcal{S}[\![f\,\texttt{(}x_1\texttt{,}\,\ldots\,\texttt{,}x_n\texttt{)}\ \texttt{=}\ e]\!] & \mid f \notin \mathrm{Dom}\,\Gamma & = & \epsilon & \text{-- unused code}\\
 & \mid \Gamma\, f \neq \top & = & \epsilon & \text{-- constant function}\\
 & \mid \text{otherwise} & = & f\ \texttt{(}x_1\texttt{,}\ \ldots\texttt{,}\ x_n\texttt{)}\ \texttt{=}\,\mathcal{S}[\![e]\!] & \\
\mathcal{S}[\![\texttt{letrec}\ f_1,\ldots,f_2\ \texttt{in}\ e]\!] & & = & & \\
\quad \texttt{letrec}\ \mathcal{S}[\![f_1]\!],\ \ldots,\ \mathcal{S}[\![f_2]\!]\ \texttt{in}\ \mathcal{S}[\![e]\!] & & & & \\
\end{array}
$$

3.3 Side-Effects

So far, we have not discussed how $\mathsf{Scc}_{\text{ANF}}$ treats side-effecting primitives (such as I/O operations). They require some extra care, but do not pose any fundamental problems. Whenever $\mathcal{A}$ encounters a side-effecting primitive p (second equation in Figure 4), the result of evaluating the application of p is assumed to be $\top$. Moreover, any function whose body contains a side-effecting primitive is mapped to $\top$ in Γ. The rationale for the latter is that, even if the function is constant, it must not be removed as its invocation carries an effect.

Note that the existence of side-effecting primitives is the only reason why we need to be able to distinguish between functions that are dead code and non-terminating functions. If a non-terminating function has an effect, we cannot replace it by an arbitrary diverging computation. In other words, the functions

$$
\begin{array}{lcl}
\mathit{undefined}\ \texttt{()} & \texttt{=} & \mathit{undefined}\ \texttt{()}\\
\mathit{ones}\ \texttt{()} & \texttt{=} & \texttt{let}\ x\ \texttt{=}\ \mathit{write\_char}\ \texttt{('1')}\ \texttt{in}\ \mathit{ones}\ \texttt{()}
\end{array}
$$

need to be treated differently; although, $\mathcal{A}$ would assign $\bot$ to both.

3.4 Inlining

As ANF already includes a notion of function abstraction, it lends itself to a uniform representation of a set of functions or procedures, which opens a path to inter-procedural analysis and, in our case, inter-procedural constant propagation. It is well known that combining inlining or procedure integration with constant propagation improves the results. For non-recursive functions (that do not contain side-effecting primitives), this can be easily achieved in $\mathsf{Scc}_{\mathrm{ANF}}$, by treating them as primitives in $\mathcal{A}$ and $\mathcal{S}$.

More precisely, in the second equation of the definition of both $\mathcal{A}$ and $\mathcal{S}$, we replace the condition $f \in \mathsf{Prim}$ by a more liberal condition of the form "$f \in \mathsf{Prim}$ or f is non-recursive." Moreover, we extend $\mathcal{E}$ and $\mathcal{E}_{\mathsf{Abs}}$ by the ability to evaluate non-recursive functions.

3.5 Higher Order Functions

$\mathsf{Scc}_{\mathrm{ANF}}$ requires a first-order program, as it does not explicitly handle higher-order variables. Nevertheless, the treatment of higher-order variables is important for modern object-oriented and functional languages. There are various ways—of varying sophistication— in which $\mathsf{Scc}_{\mathrm{ANF}}$ can be extended to handle higher-order variables. The simplest scheme, giving the least precise results, refines Γ to $\Gamma \sqcap [x_1 \mapsto \top, \ldots, x_n \mapsto \top]$ for all functions $f(x_1, \ldots, x_n) = e$ that are assigned to a variable (as opposed to being invoked). This reflects that we cannot gain any knowledge about the range of argument values at which f is invoked. Moreover, $\mathcal{A}$ needs to return $\top$ for any higher-order call site $x(x_1, \ldots, x_n)$, where x is an unknown function value.

Although the above mapping of the x_i to $\top$ is correct, it is too conservative as we may be able to statically infer the call sites of some functions that are assigned to variables. In particular, we improve on the basic scheme by using constant propagation to identify higher-order call sites that, for all possible executions of a program, invoke the same function. To do so, it suffices to extend the set of constants contained in the abstract domain Abs with symbols representing the various functions that are used as values. Whenever $\mathcal{A}$ derives $\Gamma x = f$ for a call site $x(x_1, \ldots, x_n)$, we can treat this call site as $f(x_1, \ldots, x_n)$.

However, by noting the similarity of $\mathsf{Scc}_{\mathrm{ANF}}$ and *abstract interpretation* [6] as well as the fact that $\mathsf{Scc}_{\mathrm{ANF}}$ corresponds to what is known as a 0CFA analysis [19], we can use power sets of abstract closures as the abstract domain for higher-order variables. A detailed description of the resulting algorithm is beyond the scope of this paper, but it is related to the constant propagation algorithm studied by Sabry and Felleisen [17].

4 Soundness of the New Algorithm

To establish soundness, we need three auxiliary definitions. The first one relates environments w.r.t. the free variables of an expression. It is oblivious to the entries relating to bound variables.

Definition 4.1 $\Gamma_0 \leq_{\mathrm{FV}(e)} \Gamma_1 \Leftrightarrow \forall\, x \in \mathrm{FV}\ e \cap \mathrm{Dom}\ \Gamma_0, \Gamma_0\, x = \Gamma_1\, x$.

The next two definitions cover the validity of the algorithm $\mathcal{A}$ and its auxiliary function $\mathcal{A}_{\mathsf{fix}}$ for one particular expression.

Definition 4.2 $\mathcal{A}$ is *valid* for an expression e (denoted $\mathcal{A}[\![e]\!]$) iff for any two environments Γ_0 and Γ', with $\Gamma_0 \leq_{\mathrm{FV}(e)} \Gamma'$, $\mathcal{A}[\![e]\!]\,\Gamma_0\,\{\} = \langle a,\ \Gamma,\ \Omega\rangle$ implies $\mathcal{E}_{\mathsf{Abs}}[\![e]\!]\,\Gamma' = \mathcal{E}_{\mathsf{Abs}}[\![\mathcal{S}\ e\ \Gamma]\!]\,\Gamma'$; furthermore, if $\Omega = \{\}$, $\mathcal{E}_{\mathsf{Abs}}[\![e]\!]\,\Gamma' = a$.

Definition 4.3 $\mathcal{A}_{\mathsf{fix}}$ is *valid* for a set of function bindings *fs* and a work list Ω_0 (denoted $\mathcal{A}_{\mathsf{fix}}[\![fs]\!]\,\Omega_0$) iff for any two environments Γ_0 and Γ', with $\Gamma_0 \leq_{\mathrm{FV}(fs)} \Gamma'$, we have for every binding $f\ (x_1, \ldots, x_n) = e$, $\mathcal{A}_{\mathsf{fix}}[\![fs]\!]\,\Gamma_0\,\Omega_0 = \langle\Gamma,\ \Omega\rangle$ implies $\mathcal{E}_{\mathsf{Abs}}[\![e]\!]\,\Gamma' = \mathcal{E}_{\mathsf{Abs}}[\![\mathcal{S}\ e\ \Gamma]\!]\,\Gamma'$ whenever $f \in \Omega_0$.

Since the relation $\leq_{\mathrm{FV}(e)}$ is oblivious to bound variables, validity implies that the algorithm yields a suitable environment even if bound variables in the input environment differ from the actual value assigned to a variable. This property is important as the algorithm is optimistic—i.e., intermediate environments constructed during execution of the algorithm may make incorrect assumptions about the values of some variables. However, the algorithm does not terminate unless all of these bindings are replaced by $\top$.

Theorem 4.4 (Soundness of $\mathcal{A}$) *$\mathcal{A}$ is valid for all ANF expressions e.*

The proof proceeds by induction over the nesting depth of `letrec`s in an expression and contains the following three main steps:

(i) **Base Case ($\mathcal{A}(0)$)**: We assert that for every ANF expression e that contains no `letrec` expression, we have $\mathcal{A}[\![e]\!]$.

(ii) **Auxiliary Step ($\mathcal{A}(n) \Rightarrow \mathcal{A}_{\mathsf{fix}}(n)$)**: Given a list of function bindings *fs* and a work list Ω, we assert that $\mathcal{A}_{\mathsf{fix}}[\![fs]\!]\,\Omega$ holds only if, for all bindings $f\ (x_1, \ldots, x_n) = e$ from *fs*, $\mathcal{A}[\![e]\!]$ holds.

(iii) **Inductive Step ($\mathcal{A}(n) \Rightarrow \mathcal{A}(n+1)$)**: Given a list of function bindings *fs* and an ANF expression e, we assert that $\mathcal{A}[\![e]\!]$ and $\mathcal{A}_{\mathsf{fix}}[\![fs]\!]$ together imply $\mathcal{A}[\![$`letrec` *fs* `in` $e]\!]$.

We detail these three steps in the unabridged version of this paper [5].

5 Conclusion

We have formalised the mapping of programs in SSA form to ANF and presented an ANF version of Wegman & Zadeck's conditional constant propagation algorithm, which we called $\mathsf{Scc}_{\mathrm{ANF}}$. Moreover, we have outlined how the ANF-based algorithm can be extended to include inlining and higher-order functions.

We are interested in assessing the usefulness of typed, functional intermediate languages in compilers for conventional languages. The algorithm

$\mathsf{Scc}_{\text{ANF}}$ is more rigorously defined than Wegman & Zadeck's original algorithm as, firstly, ANF has a well-defined semantics and, secondly, our notation is essentially a nicely typeset version of the programming language Haskell.[4] Such semantic clarity is a prerequisite for any rigorous formal reasoning about compiler optimisations. Moreover, it is a prerequisite for a serious comparison of the work on abstract interpretation with that in the classic dataflow analysis literature. An alternative method for formalising conditional constant propagation in a graph-based framework was introduced by Lerner et al. [13]. However, their focus is on the modular composition of dataflow analyses.

Acknowledgements. We thank the referees for their helpful comments.

References

[1] Alpern, B., M. N. Wegman and F. K. Zadeck, *Detecting equality of variables in programs*, in: *15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1998), pp. 1–11.

[2] Appel, A. W., "Modern Compiler Implementation in ML," Cambridge University Press, 1998.

[3] Appel, A. W., *SSA is functional programming*, ACM SIGPLAN Notices **33** (1998), pp. 17–20.

[4] Ballance, R. A., A. B. Maccabe and K. J. Ottenstein, *The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages*, in: *Proceedings of the Conference on Programming Language Design and Implementation* (1990), pp. 257–271.

[5] Chakravarty, M. M. T., G. Keller and P. Zadarnowski, *A functional perspective on SSA optimisation algorithms—unabridged*, Technical Report 0217, University of New South Wales (2003).

[6] Cousot, P. and R. Cousot, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, in: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (1977), pp. 238–252.

[7] Cytron, R., J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck, *Efficiently computing static single assignment form and the control dependence graph*, ACM Transactions on Programming Languages and Systems **13** (1991), pp. 451–490.

[8] Flanagan, C., A. Sabry, B. F. Duba and M. Felleisen, *The essence of compiling with continuations*, in: *Proceedings ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation, PLDI'93* (1993), pp. 237–247.

---

[4] A concrete implementation of the two functions $\mathcal{A}$ and $\mathcal{S}$, in Haskell, is available from `http://www.cse.unsw.edu.au/~patrykz/ssa-lambda/`.

[9] Harper, R. and G. Morrisett, *Compiling polymorphism using intensional type analysis*, in: *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1995), pp. 130–141.

[10] Johnsson, T., *Lambda lifting: Transforming programs to recursive equations*, in: Jouannaud, editor, *Proceedings of the International Conference on Functional Programming and Computer Architecture*, number 201 in Lecture Notes in Computer Science (1985).

[11] Jones, S. P. and A. L. M. Santos, *A transformation-based optimiser for Haskell*, Science of Computer Programming **32** (1998), pp. 3–47.

[12] Kelsey, R. A., *A correspondence between continuation passing style and static single assignment form*, ACM SIGPLAN Notices **30** (1995), pp. 13–22.

[13] Lerner, S., D. Grove and C. Chambers, *Composing dataflow analyses and transformations*, in: *Proceedings of the 29th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (2002), pp. 270–282.

[14] Morrisett, G., D. Walker, K. Crary and N. Glew, *From System F to typed assembly language*, ACM Transactions on Programming Languages and Systems **21** (1999), pp. 528–569.

[15] Peyton Jones, S. et al., *Haskell 98: A non-strict, purely functional language* (1999).
URL `http://haskell.org/definition/`

[16] Peyton Jones, S. L., *Compiling Haskell by program transformation: a report from the trenches*, in: H. R. Nielson, editor, *Proceedings of the European Symposium on Programming*, Lecture Notes in Computer Science **1058** (1996), pp. 18–44.

[17] Sabry, A. and M. Felleisen, *Is continuation-passing useful for data flow analysis?*, in: *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)* (1994), pp. 1–12.

[18] Schneider, F. B., G. Morrisett and R. Harper, *A language-based approach to security*, in: *Informatics: 10 Years Back, 10 Years Ahead*, Lecture Notes in Computer Science **2000** (2000), pp. 86–101.

[19] Shivers, O., *Control-flow analysis in Scheme*, in: *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation* (1988).

[20] Tarditi, D., G. Morrisett, P. Cheng, C. Stone, R. Harper and P. Lee, *TIL: A type-directed optimizing compiler for ML*, in: *SIGPLAN Conference on Programming Language Design and Implementation*, 1996, pp. 181–192.

[21] Tolmach, A. et al., *An external representation for the GHC core language* (2001).

[22] Wegman, M. N. and F. K. Zadeck, *Constant propagation with conditional branches*, ACM Transactions on Programming Languages and Systems **13** (1991), pp. 181–210.