Horn Clauses as an Intermediate Representation for Program Analysis and Transformation∗

GRAEME GANGE
Department of Computing and Information Systems
The University of Melbourne, Victoria 3010, Australia
(*e-mail:* gkgange@unimelb.edu.au)

JORGE A. NAVAS
NASA Ames Research Center, Moffet Field CA
(*e-mail:* jorge.a.navaslaserna@nasa.gov)

PETER SCHACHTE, HARALD SØNDERGAARD, PETER J. STUCKEY
Department of Computing and Information Systems
The University of Melbourne, Victoria 3010, Australia
(*e-mail:* {schachte,harald,pstuckey}@unimelb.edu.au)

Abstract

Many recent analyses for conventional imperative programs begin by transforming programs into logic programs, capitalising on existing LP analyses and simple LP semantics. We propose using logic programs as an intermediate program representation throughout the compilation process. With restrictions ensuring determinism and single-modedness, a logic program can easily be transformed to machine language or other low-level language, while maintaining the simple semantics that makes it suitable as a language for program analysis and transformation. We present a simple LP language that enforces determinism and single-modedness, and show that it makes a convenient program representation for analysis and transformation.

KEYWORDS: compilers, control flow graphs, intermediate representation, program analysis and transformation, SSA

1 Introduction

Most compilers, regardless of the programming language(s) and paradigms supported, use some *Intermediate Representation* (IR) between parsing the input program and emitting the object code. Use of an IR has the significant advantage of allowing a compiler to target multiple CPU architectures, and even multiple programming languages, without duplicating the bulk of the compiler, which operates exclusively on the IR. Over the course of the compilation, this representation will be analysed for different characteristics and transformed in various semantics-preserving ways, in preparation for efficient object code generation. Thus it is important for an IR to make program analysis and transformation as simple and convenient as possible. *Three-address code* has been a popular form

∗ This work was supported by the Australian Research Council through Discovery Project Grant DP140102194.

$$
\begin{array}{lcl@{\qquad}lcl}
\mathit{Prog} & \to & \mathit{Func}^* & \mathit{Head} & \to & \mathit{Name}(\mathit{Var}^*) \\
\mathit{Func} & \to & \mathit{Head}\ \mathit{Block}\ \mathit{Block}^* & \mathit{Prim} & \to & \mathit{Var} = \mathit{Val} \\
\mathit{Block} & \to & \mathit{BlockID} : \mathit{Prim}^*\, \mathit{BlockExit} & & | & \mathit{Var} = \mathit{Val} \odot \mathit{Val} \\
\mathit{BlockExit} & \to & \textbf{return}\ \mathit{Val} & & | & \mathit{Name}(\mathit{Val}^*) \\
 & | & \textbf{if}\ \mathit{Test}\ \mathit{BlockID}\ \mathit{BlockID} & \mathit{Test} & \to & \mathit{Val} \oslash \mathit{Val} \\
 & | & \textbf{goto}\ \mathit{BlockID} & \mathit{Val} & \to & \mathit{Var} \mid \mathit{Const}
\end{array}
$$

Fig. 1. A three-address code language

$$
\begin{array}{lcl}
\mathit{Block} & \to & \mathit{BlockID} : \mathit{Phi}^*\, \mathit{Prim}^*\, \mathit{BlockExit} \\
\mathit{Phi} & \to & \mathit{Var} = \varphi(\mathit{Var}^*)
\end{array}
$$

Fig. 2. Changes to three-address language to produce SSA

for this purpose for many years. Figure 1 presents a three-address code language. Here we assume we are given *Name*, the set of all possible function names; *Var*, the set of variable names; and *Const*, the set of all primitive constant values. We let $\mathit{Primval} = \mathit{Var} \cup \mathit{Const}$. To simplify exposition, we let $\odot$ stand for all primitive arithmetic and logical operators, and $\oslash$ stand for all primitive binary comparison operators.

Each *basic block* of a function is a sequence of function calls and primitive instructions, ending with a control transfer to another basic block. Once control enters a basic block, it is guaranteed to reach its end (unless some exceptional circumstance arises). This guarantee makes analysis of each basic block straightforward.

A popular variant of three-address code is *Static Single Assignment* (SSA) form (Alpern et al. 1988; Cytron et al. 1991; Lattner and Adve 2004). SSA was proposed as a way to generalize *value numbering*, a technique used to remove redundant computation. In SSA form, each variable is assigned at most once in its scope. Where a variable would be re-assigned, a new variable is instead introduced. Since each variable is only assigned once, it is not necessary to consider the program point when referring to a variable, only the function it appears in. This makes many analyses simpler and more efficient, because a single abstract value can be associated with each variable name in a function, and the set of variable names of interest is limited and easily determined.

A basic block with multiple predecessors presents a complication for SSA: a variable use in such a block may refer to definitions of those variables in any of the predecessor blocks. To give such a variable a single definition, SSA introduces the concept of a φ *node*: the variable is assigned the result of a "fake" function that takes as input the version of the variable from each predecessor block. A block with multiple predecessors will contain as many φ nodes as it has variables with alternative definitions in earlier blocks. Figure 2 presents the changes to three-address syntax needed to transform to SSA: each block may begin with φ nodes. Consider, for example, the C code to compute the greatest common divisor shown in Figure 3 (left side). This code can be converted into SSA form as shown in Figure 3 (right).

Several researchers have presented program analyses that work by first transforming an imperative source program (*e.g.*, Spoto et al. (2010) and Albert et al. (2012)), or Java bytecode (*e.g.*, Benton and Fischer (2007)) into an abstract form based on the constructs of logic programming, and then analysing this result. Others (*e.g.*, Whaley et al. (2005)) have used logic programs to represent program analyses. In some cases this

```
int gcd(int a, int b) {
  while (b != 0) {
    int t = b;
    b = a % t;
    a = t;
  }
  return a;
}
```

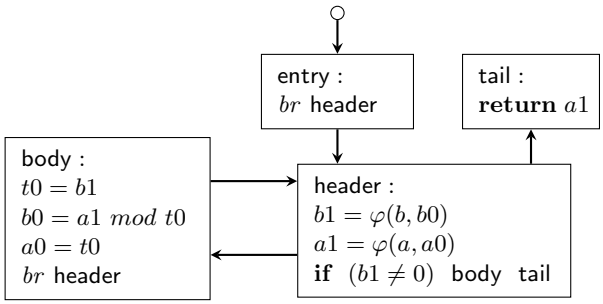

Fig. 3. The `gcd` function in C (left) and LLVM-style SSA form (right)

benefits from existing logic program analyses, but the greater benefit derives from the simple, traditional T_P semantics for logic programs, and hence simpler and more powerful analyses. Logic programs have none of the limitations of SSA form that we detail below.

In this paper we propose representing an imperative source program as a logic program throughout the compilation process. It may be surprising to think of compiling C programs by translation to Prolog, rather than the reverse, but we show that placing a few limitations on the generated logic programs leaves low-level programs suitable for high-level analysis and transformation, and also for final translation to machine language.

In Section 2 we discuss problematic aspects of SSA and related forms, together with suggested ways of addressing the problem. In Section 3 we introduce "Logic Programming (LP) Form" and we show how to translate a three-address code to it. In Sections 4 and 5 we give example analyses for LP form. In Section 6 we discuss related work. Finally, Section 7 reviews what has been achieved with the proposed LP form, and concludes.

2 SSA and Allied Forms: Problems and Solutions

While SSA form does simplify a number of common program analyses, it has significant limitations that interfere with others. Most of these problems can be solved, at the cost of further complicating the SSA form. In this section we will consider these limitations.

2.1 Path obliviousness

Basic blocks do not indicate the constraints that must be satisfied for them to be entered. These constraints appear in predecessor blocks. In a forward analysis, this means constraints must be propagated from conditional branches to their target blocks. A backward analysis is clumsier: it must peek backward into each predecessor block to see what conditions hold.

Consider, for example, forward interval analysis of the code shown in Figure 4. The blocks left and right both refer to "x" so there is no straightforward way to

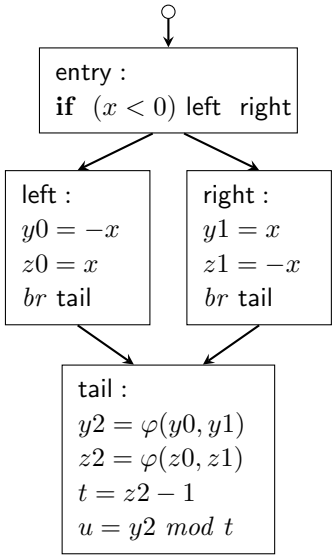

Fig. 4. SSA and branching

separate the reasoning that needs to be done under different assumptions about x. The next section considers the use of *different* names for x in the separate branches, which would help in this example. However, as it stands, we cannot assign non-trivial intervals to $y2$ and $z2$ in the absence of path constraints that record how control reached tail.

To solve the path obliviousness problem, Ballance et al. (1990) have proposed the use of *Gated Single-Assignment* form (GSA). SSA's φ nodes are replaced by different types of *gating functions*. These capture the control conditions that determine which of the various definitions that reach the node should provide its value. One gating function, γ, is in essence an if-then-else function. For example we might translate $\varphi(x_1, x_2)$ to $\gamma(P, x_1, x_2)$, where P is some branch condition from elsewhere in the program. Flow of definitions inside loops are managed by additional gating functions to handle initial and loop-carried values (μ nodes) as well as loop-exiting values (η nodes).

This form makes information flow more transparent, but it is extremely complex, compared to SSA. The form we propose has greater uniformity, as it does not introduce a variety of different mechanisms for the "joining" or "merging" of information. Moreover, GSA, as SSA, does not readily lend itself to backward analysis, as discussed next.

2.2 Forward bias

The φ nodes of SSA are convenient when analysing each basic block, as they clearly indicate which variables of which other blocks provide values to the variables of the block. However, this assumes *forward* analysis. In this direction, where execution paths join, each variable with alternative sources is indicated by a φ node specifying the different names for each alternative, and because each variable is only defined once per function, it naturally receives only one abstract value during forward analysis.

For a backward analysis, however, there is no node dual to a φ to indicate the alternative destinations that may use each variable following a branch instruction. (While the branch indicates alternative destinations, it does not specify the variables that may be used there). Importantly, the alternative destinations for a branch all have the same name for each variable. In a backward analysis, then, different blocks of a function may determine different abstract values for the same variable name: the virtue of SSA that each variable has a unique definition in each function does not apply to backward analysis.

Consider Figure 4 and suppose we wish to verify whether the division is safe, *i.e.* that t cannot be 0. In a fixed-width integer context (as we assume here), it is convenient to use "wrapped intervals" (Gange et al. 2015) as an abstract domain, as these allow us to capture both intervals and complemented intervals. Reasoning backwards, we find the following sufficient safety condition for the tail block: $z0, z1 \notin [1, 1]$. For x, this then translates to $x \notin [1, 1]$ (for left), and $x \notin [-1, -1]$ (for right). This allows us to conclude that all will be well if $x \notin [-1, 1]$, but that is insufficient to prove safety.

To address this problem, Ananian (1999) has proposed adding σ nodes to SSA form to create *Static Single Information* (SSI) form. Where SSA form has a φ node at the top of each block indicating where the value of each variable in the block comes from, SSI adds a σ node at the bottom of each branching block indicating where each variable's value goes to. This permits reasoning in both directions and provides (variable) names for all relevant pieces of information. However, it does not address a number of other problems, as we now explain.

2.3 Lack of φ node compositionality

For a non-relational *value analysis*, which assigns each variable a single abstract value, a φ node conveniently specifies that the abstract value for a variable is the join of the abstract values of the input variables. For a relational analysis, however, a φ node does not have such a simple interpretation. Consider an octagon analysis (Miné 2006) of the program snippet in Figure 4. For the two transitions to the tail block, we have $x - z0 = 0 \wedge y0 + z0 = 0$ and $x - y1 = 0 \wedge y1 + z1 = 0$. Or, assuming SSI, we have $x0 \geq 0 \wedge x0 - z0 = 0 \wedge y0 + z0 = 0$ and $x1 < 0 \wedge x1 - y1 = 0 \wedge y1 + z1 = 0$. In either case, there is no meaningful (abstract) interpretation of the statement $y2 = \varphi(y0, y1)$ in isolation that does not throw away most of these relationships. In particular, we lose the fact that $y2 + z2 = 0$. The two φ nodes must be treated together to prevent this loss of precision. What is really needed is a single node that conveys the information of $(y2, z2) = \varphi((y0, z0), (y1, z1))$.

SSI does not help with this problem and in fact the remedies discussed so far appear to address particular symptoms rather than a more fundamental cause which, in our view, is an insufficiently abstract view of name management.

2.4 Name management

The φ and σ nodes of SSA and SSI form require special treatment during analysis. A φ node $v = \varphi(v1, v2, \ldots)$, cannot be treated like a function call, because the variables mentioned come from alternative blocks—that is, they cannot exist at the same time. For example, when analysing a basic block beginning with $v6 = \varphi(v3, v5)$, we must find the analysis results for the two predecessor blocks, rename $v3$ to $v6$ in the first and $v5$ to $v6$ in the second, and then find the join of the two and project away other variables in the originating blocks. In essence, all φ (and σ) nodes in a block must be treated similarly to the way a function call is treated: information about actual parameters must be renamed to match formal parameters (or vice-versa for backward analysis), information about variables not conveyed in the call must be projected away, and the join of all incoming calls must be taken.

Appel (1992) and Kelsey (1995) observed similarities between SSA and continuation-passing style in functional programming. Later Appel (1998) observed that SSA is in a sense equivalent to functional programming *without* continuations, and he presented a transformation from SSA to functional program (FP) form. This form mitigates the name management problem, using parameter passing to serve the purpose of φ nodes: where SSA form would have a block with a φ node for each variable defined in predecessor blocks, the FP form has a function with a parameter for each variable defined outside. Likewise, FP form uses function calls in place of jumps between blocks. Since SSA form supports function call and return in addition to φ nodes and jumps between blocks, FP form is notably simpler than SSA. So while analyses for SSA form are often only intra-procedural, analyses for FP form will naturally be inter-procedural as well.

Appel's note "SSA is functional programming" (Appel 1998) conveys these points very clearly. But a corollary is that functional form also preserves forward bias. We share the enthusiasm for a declarative formalism but we also point out that a relational view can offer greater flexibility than a functional view.

$$
\begin{array}{lcllcl}
\mathit{Prog} & \to & \mathit{Proc}^* & \mathit{Goal} & \to & \odot(\mathit{Val}^*; \mathit{Var}^*) \\
\mathit{Proc} & \to & \mathit{Clause}^* & & | & \circledcirc(\mathit{Val}, \mathit{Val};) \\
\mathit{Clause} & \to & \mathit{Head} \leftarrow \mathit{Goal}^* & & | & \mathit{Name}(\mathit{Val}^*; \mathit{Var}^*) \\
\mathit{Head} & \to & \mathit{Name}(\mathit{Var}^*; \mathit{Var}^*) & \mathit{Val} & \to & \mathit{Var} \mid \mathit{Const}
\end{array}
$$

Fig. 5. An LP form language

2.5 Input/output asymmetry

While each input parameter of a function has a unique name apparent in the function header, the return value cannot be determined without scanning all the function blocks. In fact, there may be many alternative variables returned by different blocks. This is inconvenient for any summarising analysis, which ultimately must project the analysis result for the function onto the function inputs and output. For example, in Figure 3, one knows that $a1$ and $b1$ are input to the function (this is omitted from the figure to save space), but must examine all the blocks of the function to see that $a1$ is the output. In fact, a function with more than one return may have many different output variables.

These problems can be avoided by first transforming the function replacing all return statements with jumps to a distinguished new final block containing a φ node joining all return values into a new variable, which is then returned. If the function header is augmented to record this final variable name along with the function parameters, it would not be necessary to scan all blocks to find the unique return variable. This extra step is not difficult, but is unnecessary for LP form.

A related inconvenience is the fact that functions can only return a single result. If, for example, two functions compute different values through similar computations, and the two are often called together, it may be desirable to fuse the two functions into a single one that returns two values. Of course, this may be done by returning a tuple, but in this case a structure is returned instead of two separate values, which may thwart many analyses. This can be solved by allowing functions to return multiple separate values.

2.6 Implicit variable scoping

While the φ nodes of a block indicate some of the defined variables on entry to the block, they do not indicate all of them. In fact, a block with only one predecessor will generally not have any φ nodes at all, and so no indication at all of which variables are defined on entry. Neither does SSA form provide any indication of which variables of one block are communicated to its successors. For analyses whose efficiency depend on minimising the number of variables under consideration, knowing which variables enter and leave a block would allow irrelevant variables to be projected away.

3 LP Form

SSA is a small refinement of three-address code. We argue that a larger refinement, to a restricted form of logic programming, provides the single-assignment benefits of SSA for ease of analysis while avoiding the problems outlined in Section 2.

Figure 5 presents a restricted Logic Programming language suitable for representing low-level programs.[1] In addition to fitting this grammar, LP form requires that for each guard (*i.e.*, $\Subset$) in each clause, there must be at least one other clause for the same procedure that is identical up to that guard, followed by the complementary guard, and any two clauses for a procedure must contain complementary guards, up to which they are identical.[2] Furthermore, all clause heads for a given procedure must be identical. This tames the nondeterminism of logic programming, ensuring that exactly one clause will succeed for each set of inputs, and makes analysis of procedures with multiple clauses simpler. That is, only one clause of each procedure will be executed, and no backtracking will be necessary.

This form also tames the multiple modes of a logic program by explicitly dividing the arguments into inputs followed by outputs, separated by a semicolon. In calls to primitive as well as user defined procedures, an input argument must be either a variable or a constant value, and an output argument must be a variable. All parameters in procedure heads must be variables. This ensures that variables are free until they are assigned, after which they are ground. As in Mercury (Somogyi et al. 1996), no dereferencing is ever needed.

LP form differs from SSA form in the following ways:

- Instead of blocks, LP form has *clauses*; a procedure comprises one or more clauses, exactly one of which will be executed.
- Instead of conditional constructs and computed jumps, LP form has *guards*, instructions that can either succeed or fail, determining which clause will be executed.
- It replaces unconditional branches with procedure calls, and loops with recursion.
- All registers (variables) in a clause are either parameters to that procedure or are defined in that clause, thus it has no need for φ nodes.
- It uses parameters to pass data out of, as well as into, procedures, thus it has no return instruction.
- It explicitly models changes to data structures and input/output operations, allowing pure functions to be recognised and optimised. SSA could do this, but, at least in the LLVM implementation, does not.
- Where SSA form has four different control transfer operations, plus φ nodes, LP form has only procedure calls and multiple clauses, so LP form is simpler.

One disadvantage of this representation is that the common initial parts of the clauses are duplicated for each clause, leading to duplicated analysis effort. Our current preliminary implementation factors out the duplicated code, representing a procedure body as a tree, with a sequence of goals at each node, and optionally a guard and two child nodes. This not only avoids duplicated analysis work, but also ensures that the clauses remain mutually exclusive and exhaustive through any program transformations.

[1] Details such as handling of type information and symbol tables are outside the scope of this paper. Our handling of them is similar to that of other IRs.

[2] Note that, in LP form constructed directly from three-address code, there will be at most one guard in a clause; however, inlining can produce clauses with multiple guards.

$$\frac{\begin{array}{c} v = \mathsf{vars}(B_0, ..B_n) \quad \langle B_0, \overline{v}, \mathsf{id}\rangle \Rightarrow \langle B_0', C_0, \theta_0\rangle \quad \cdots \quad \langle B_n, \overline{v}, \mathsf{id}\rangle \Rightarrow \langle B_n', C_n, \theta_n\rangle \\ H = f(\overline{p}, \mathsf{st}; \mathsf{ret}, \mathsf{st}\theta_0) \quad H_0 = f_{B_0}(\overline{v}, \mathsf{st}; \mathsf{ret}, \mathsf{st}\theta_0)) \quad \cdots \quad H_n = f_{B_n}(\overline{v}, \mathsf{st}; \mathsf{ret}, \mathsf{st}\theta_n)) \end{array}}{f(\overline{p})B_0, \ldots B_n \implies (H \leftarrow H_0) \wedge (H_0 \leftarrow B_0') \wedge C_0 \wedge \cdots \wedge (H_n \leftarrow B_n') \wedge C_n}$$

$$\frac{\langle \Pi, \overline{v}, \theta\rangle \Rightarrow \langle \Phi, C_1, \theta'\rangle \quad \langle \Xi, \overline{v}, \theta'\rangle \Rightarrow \langle \Theta, C_2, \theta''\rangle}{\langle \Pi; \Xi, \overline{v}, \theta\rangle \Rightarrow \langle \Phi \wedge \Theta, C_1 \wedge C_2, \theta''\rangle} \qquad \frac{r \in \mathit{Primval} \quad \mathsf{newvar}\ v' \quad \theta' = \theta[v \mapsto v']}{\langle v = r, \overline{v}, \theta\rangle \Rightarrow \langle v' = r\theta, \mathit{true}, \theta'\rangle}$$

$$\frac{\mathsf{newvar}\ v' \quad \theta' = \theta[v \mapsto v'] \quad \overline{a'} = \overline{a}\theta}{\langle v = \odot(\overline{a}), \overline{v}, \theta\rangle \Rightarrow \langle \odot(\overline{a'}; v'), \mathit{true}, \theta'\rangle} \qquad \frac{\mathsf{newvar}\ v', \mathsf{st}' \quad \theta' = \theta[v \mapsto v', \mathsf{st} \mapsto \mathsf{st}'] \quad \overline{a'} = \overline{a}\theta}{\langle v = g(\overline{a}), \overline{v}, \theta\rangle \Rightarrow \langle g(\overline{a'}, \mathsf{st}; v', \mathsf{st}'), \mathit{true}, \theta'\rangle}$$

$$\frac{}{\langle \mathbf{return}\ v, \overline{v}, \theta\rangle \Rightarrow \langle \mathsf{ret} = v\theta, \mathit{true}, \theta\rangle} \qquad \frac{\mathsf{newvar}\ \mathsf{st}' \quad \theta' = \theta[\mathsf{st} \mapsto \mathsf{st}']}{\langle \mathbf{goto}\ B, \overline{v}, \theta\rangle \Rightarrow \langle f_B(\overline{v}, \mathsf{st}; \mathsf{ret}, \mathsf{st}'), \mathit{true}, \theta'\rangle}$$

$$\frac{\begin{array}{c} \mathsf{newproc}\ \nu \quad C_t = \nu(\overline{v}, \mathsf{st}; \mathsf{ret}, \mathsf{st}') \leftarrow v_i \circledcirc v_j \wedge f_{B_t}(\overline{v}, \mathsf{st}; \mathsf{ret}, \mathsf{st}') \\ C_f = \nu(\overline{v}, \mathsf{st}; \mathsf{ret}, \mathsf{st}') \leftarrow \neg v_i \circledcirc v_j \wedge f_{B_f}(\overline{v}, \mathsf{st}; \mathsf{ret}, \mathsf{st}') \end{array}}{\langle \mathbf{if}\ (v_i \circledcirc v_j)\ B_t\ B_f, \overline{v}, \theta\rangle \Rightarrow \langle \nu(\overline{v}, \mathsf{st}; \mathsf{ret}, \mathsf{st}'), C_t \wedge C_f, \theta\rangle}$$

Fig. 6. Translation from three-address code to LP form

3.1 Translation to LP form

To simplify exposition, we assume the source program is presented in three-address code form.[3] We denote by $\overline{v}$ a sequence of the 0 or more variables comprising the set v.

To track side-effects, our translation uses the distinguished variable st to denote the state of the computation, including the heap and input/output state. This ensures operations that may have side-effects will be executed in the correct order, while allowing pure operations to be reordered. We also use ret to hold the value returned by the function.

Figure 6 presents our translation. Here the notation $\Phi \Longrightarrow \Psi$ indicates that the function definition Φ is transformed to the clauses Ψ. In the remaining transforms, the notation $\langle \Phi, \overline{v}, \theta\rangle \Rightarrow \langle \Psi, C, \theta'\rangle$ means that, in the context of substitution θ and variables $\overline{v}$, statements Φ are translated to goals Ψ, with extra clauses C and resulting substitution θ'. The substitutions are used to ensure each variable has a single assignment, and the extra clauses are for auxiliary predicates generated to implement conditionals. We let newvar x and newproc x specify that x is a fresh variable or procedure name, respectively.

As indicated by the first transform, each basic block is transformed into a single clause procedure, with one extra clause to invoke the first. For simplicity, each of these clauses takes all the variables appearing in the function, plus the state variable st as inputs, and the return value variable ret and the state, as modified by the block body, as outputs. The final transform produces a two-clause procedure for each conditional primitive. Because these transforms are idempotent and non-overlapping, confluence is assured.

Figure 7 shows the `gcd` function of Figure 3 translated to LP form. The transformation is rather simple-minded, threading every variable to each clause. However, the neededness analysis described in Section 5 allows the removal of unnecessary variable threading,

[3] Because variables in LP form are scoped to a single clause, rather than to all the blocks of a function body, translation from SSA is actually *less* convenient than from three-address code.

$$gcd(a, b, \mathsf{st}; \mathsf{ret}, \mathsf{st}') \leftarrow gcd_{header}(a, b, t, \mathsf{st}; \mathsf{ret}, \mathsf{st}')$$

$$gcd_{header}(a, b, t, \mathsf{st}; \mathsf{ret}, \mathsf{st}') \leftarrow gcd_{\nu}(a, b, t, \mathsf{st}; \mathsf{ret}, \mathsf{st}')$$

$$gcd_{\nu}(a, b, t, \mathsf{st}; \mathsf{ret}, \mathsf{st}') \leftarrow b \neq 0 \wedge gcd_{body}(a, b, t, \mathsf{st}; \mathsf{ret}, \mathsf{st}')$$
$$gcd_{\nu}(a, b, t, \mathsf{st}; \mathsf{ret}, \mathsf{st}') \leftarrow b = 0 \wedge gcd_{tail}(a, b', t, \mathsf{st}; \mathsf{ret}, \mathsf{st}')$$

$$gcd_{body}(a, b, t, \mathsf{st}; \mathsf{ret}, \mathsf{st}') \leftarrow t' = b \wedge \mathrm{mod}(a, t'; b') \wedge a' = t' \wedge gcd_{header}(a', b', t', \mathsf{st}; \mathsf{ret}, \mathsf{st}')$$

$$gcd_{tail}(a, b, t, \mathsf{st}; \mathsf{ret}, \mathsf{st}) \leftarrow \mathsf{ret} = a$$

Fig. 7. The `gcd` program translated to LP form

$$gcd(a, b; \mathsf{ret}) \leftarrow b \neq 0 \wedge \mathrm{mod}(a, b; b') \wedge gcd(b, b'; \mathsf{ret})$$
$$gcd(a, b; \mathsf{ret}) \leftarrow b = 0 \wedge \mathsf{ret} = a$$

Fig. 8. The translated `gcd` program of Figure 7 after simplification

and a simple inlining heuristic can remove unnecessary procedures. Figure 8 shows the translated `gcd` program after these transformations.

3.2 Translation from LP form to machine language

The Mercury project (Somogyi et al. 1996) has demonstrated that logic programs can be translated to very efficient executable code by tracking predicate determinism at compile-time and eliminating variable dereferencing. LP form likewise eschews unification of "logic variables" and the need for dereferencing, but goes further, eliminating nondeterminism and the need for choicepoints and a machine register to track them. Since LP form is designed to be suitable for any language, it does not provide its own memory management solution, and so does not need a register to control memory allocation.

In fact, LP form is surprisingly close to the machine language of common computers. Its ability to express operations with multiple outputs better reflects CPU capabilities than the functional restriction imposed by common three-address languages. For example, the x86 architecture's `IDIV` instruction produces both a quotient and a remainder in separate registers, and numerous instructions modify flags in addition to other registers; these are better abstracted in LP form than in three-address code.

As mentioned above, our implementation actually factors out the common initial part of all the clauses for a procedure. That is, each procedure is represented as a body, which is a list of goals optionally ending with a test to select between two (or more) subsequent bodies. This representation closely matches the structure of the code to be generated: some straight-line code ending with a conditional branch to one alternative and a fall through to the other.

The end of each clause is also easily translated through last call optimisation: if the final operation in a clause is a procedure call, that call is changed to an unconditional branch to the destination. If it is a primitive, it is followed by a return instruction. Other than this, machine code generation for LP form is similar to SSA or three-address code.

$$p(x,u) \leftarrow x < 0 \wedge negate(x,y) \wedge z = x \wedge p_1(y,z,u)$$
$$p(x,u) \leftarrow x \geq 0 \wedge y = x \wedge negate(x,z) \wedge p_1(y,z,u)$$

$$p_1(y,z,u) \leftarrow sub(z,1,t) \wedge mod(y,t,u)$$

Fig. 9. Example of Fig 4 in LP form

4 LP form analysis and transformation

In this section we show that LP form does not share the flaws discussed in Section 2, and discuss its other benefits. Consider again the example program of Figure 4. After simplification through inlining of simple procedures and elimination of unnecessary dataflow, this would be expressed in LP form as shown in Figure 9.[4] When performing a forward interval analysis on this code, the $x < 0$ condition in the first clause gives the interval $[-\infty, -1]$ for x, $[1, \infty]$ for y, and $[-\infty, -1]$ for z prior to the call to p_1. For the second clause, we infer $[0, \infty]$ for x, $[0, \infty]$ for y, and $[-\infty, 0]$ for z. Computing the join of the abstract states for the two calls to p_1, we have $y \in [0, \infty] \wedge z \in [-\infty, 0]$, so analysing p_1 gives us $y \in [0, \infty] \wedge z \in [-\infty, 0] \wedge t \in [-\infty, -1]$ on reaching the first call to *mod*, allowing us to certify the safety of the mod operation. The path-awareness of LP form gives us stronger analysis results without any extra effort.

Since each LP form clause is logically an unordered conjunction, it is equally adept at forward and backward analysis. Consider a backward analysis of the program of Figure 9 to determine the safety of modulo (division) operations. This will start with the constraint $t \neq 0$ at the end of p_1, which implies $z \neq 1$ on entry to p_1. Analysing the first clause of p backwards from its call to p_1, we deduce $z \neq 1 \vee x \neq 1 \vee y \neq -1$ before reaching the $x < 0$ goal. Handling this goal gives us $x < 0 \rightarrow z \neq 1 \vee x \neq 1 \vee y \neq -1 \equiv \mathit{True}$, meaning we have nothing else to prove for that clause. Turning to the second clause of p, we derive $x \geq 0 \rightarrow z \neq 1 \vee x \neq -1 \vee y \neq -1 \equiv \mathit{True}$, and again the proof obligation is discharged.

Relational analyses do not present any difficulty for LP form, because it has no artificial φ nodes to separately combine alternative versions of variables. This is handled through conventional procedure calls, where the least upper bound is used to combine results for multiple calls. Consider an octagon analysis (Miné 2006) of Figure 9. Much like the analysis discussed in Section 2.3, analysis derives $y + x = 0 \wedge z - x = 0 \wedge y + z = 0$ leading to the call to p_1 from clause 1, and $y - x = 0 \wedge x + z = 0 \wedge y + z = 0$ for clause 2. Procedure calls are handled by projecting the abstract state onto the variables appearing in the call, and computing the least upper bound of the states. In this case, this yields $y + z = 0 \sqcup y + z = 0 \equiv y + z = 0$, preserving the strong results obtained for both clauses.

The other issues for SSA and FP form discussed in Section 2 are trivially addressed by LP form. Lacking φ nodes, LP form has no issue with name management. Because LP form is relational, it has no issue with input/output asymmetry. And because each clause has its own scope, the scope of each variable is obvious.

[4] Since the definition of p_1 is so simple, in practice it would be inlined, but that would only give us stronger analysis results.

5 Specialised analyses for LP form

Liveness analysis is a standard program analysis used to determine for each program point the set of variables whose values may be needed later. The single assignment property enjoyed by SSA, FP, and LP forms somewhat simplifies this analysis: because each variable is assigned only once, it is not necessary to take account of variable reassignment. Within a single block (clause) of SSA (LP form) code, this is easily done by traversing the statements backward, noting the first encountered use of each variable, which will be the last use on forward execution, and each variable assignment, which will be the definition of that variable. To handle liveness for a whole function, analysis results must be propagated backward between blocks.

Dead code elimination is a transformation to remove unnecessary code. Any code that assigns only dead variables can be removed, but doing so may remove variable uses, and produce stronger results for liveness analysis. Thus it is beneficial to perform liveness analysis and dead code elimination simultaneously. If this is extended beyond individual functions to an entire module or even a whole program, more dead code can be eliminated.

We present a two-phase interprocedural *neededness* analysis, which combines liveness and dead code elimination. The first phase computes *neededness dependencies*, conjunctions of implications of the form $x \to y$ signifying that if variable x is needed on completion of a goal, then y is needed on entry. This analysis can be performed bottom-up over a module's call graph, one strongly connected component (SCC) at a time, which ensures that all callers of a given procedure, except those in the same SCC, will be analyzed before the procedure itself. A fixed point must be computed for each SCC, but no iteration is necessary between SCCs. This reduces the number of procedures analyzed in each fixed point iteration, since SCCs are typically fairly small.

Formally, we define our neededness dependency domain N as the set of conjunctions of variable $\to$ variable implications, where an individual implication $x \to y$ indicates that if variable x is needed, then so is y. We let S denote the $\mathit{Goal} \longrightarrow N$ *neededness dictionary* function space, specifying neededness dependencies for many procedures. We define our analysis with the following functions:

$$P_d :: \mathcal{P}(\mathit{Proc}) \longrightarrow S \qquad C_d :: \mathit{Goal}^* \longrightarrow S \longrightarrow N$$
$$D_d :: \mathit{Proc} \longrightarrow S \qquad G_d :: \mathit{Goal} \longrightarrow \mathcal{P}(\mathit{Var}) \longrightarrow S \longrightarrow N$$

Here P_d gives the neededness dictionary for all the procedures in the module; D_d yields the dictionary for a single procedure; C_d produces the neededness of a single clause given a neededness dictionary; and G_d gives the neededness of a single goal given the set of variables needed later in the clause body and a neededness dictionary.

As shown in Figure 10, the neededness analysis of a module is the least fixed point of the combination of results for all procedures in the module, and the result for a procedure is just the conjunction of the neededness of all its clauses, which is the conjunction of results for all goals in each clause. The analysis result for a primitive operation is the conjunction of $x \to y$ implications for each output x and each input y. For a primitive comparison operation, it is the conjunction of $x \to y$ for each variable x defined later in the clause (determined by the defs function) and each input y of the comparison. Since primitive comparisons are guards, they are only needed to determine if the following code is executed, so they are only needed if some variable defined later is needed.

$$P_d\ S = lfp\left(\bigsqcup_{d\in S} D_d\ d\right)$$

$$D_d\ [p(\overline{v_i};\overline{v_o}) \leftarrow (B_1, \vee\cdots\vee B_n)]\ A = \lambda p(\overline{v_i};\overline{v_o})\ .\ \left(\exists(\mathit{Var}\setminus v_i\setminus v_o)\ .\ \bigwedge_{1\le k\le n} C_d\ B_k\ A\right)$$

$$C_d\ (g_1\wedge\cdots\wedge g_n)\ A = \bigwedge_{1\le k\le n} G_d\ g_k\ \mathsf{defs}(g_{k+1}\wedge\cdots\wedge g_n)\ A$$

$$G_d\ \odot(\overline{v_i};\overline{v_o})\ V\ A = \bigwedge_{x\in v_i}\bigwedge_{y\in v_o} y\to x$$

$$G_d\ \circledcirc(x,y)\ V\ A = \bigwedge_{v\in V}(v\to x\wedge v\to y)$$

$$G_d\ p(\overline{v_i};\overline{v_o})\ V\ A = A\ p(\overline{v_i};\overline{v_o})$$

Fig. 10. Neededness abstract interpretation

The second analysis phase uses these dependencies to determine which procedure inputs and outputs are actually used, beginning by marking all parameters of public (exported) functions as needed. This analysis then proceeds top-down by SCCs through the program call graph, with each SCC processed until a fixed point is reached. In each iteration, each clause in the SCC is processed with a needed variable formula, initially the conjunction of the set of output variables of that procedure that are marked as needed.

Processing of a clause proceeds from last goal to first. If any output of a goal is in the needed variable formula, the goal is marked as needed, and the called procedure has its needed outputs marked for when *it* is processed. Then the neededness dictionary for the called procedure is conjoined with the current needed variable formula, and the goal's output variables are projected out, to produce the new needed variable formula. This formula then comprises the live variable set for that goal. Primitive goals are simpler: if any output is in the needed variable formula, it is marked as needed, its inputs are added to the needed variable formula, and its outputs are projected out. Once a fixed point has been reached, any goal or parameter not marked as needed can be removed.

6 Related Work

Many variants of SSA have been proposed (Ballance et al. 1990; Gerlek et al. 1995; Chow et al. 1996; Ananian 1999) and much work has been concerned with how to generate (compact) SSA and its variants efficiently (Cytron et al. 1991; Tu and Padua 1995; Ananian 1999). In Section 2 we mentioned the work on FP form by Kelsey (1995) and Appel (1998). Appel (1998) in fact sketches two translations to FP form, one producing a "flat" sequence of function definitions, the other producing nested definitions. The latter uses fewer functions and variables and Appel points out that the structure of function nesting makes the dominance properties of the original control-flow graph explicit. Appel (1998) also uses the equivalent of SSI's "σ nodes" as a pedagogic tool; the σ nodes are in fact pushed into successor blocks and become mere "renaming" φ nodes.

Peralta and Cruz-Carlón (2006) briefly sketched a translation from SSA to CLP, but

provided no formal definition of the translation. From their examples it is clear that the translation differs from the one suggested here. Peralta et al. (1998) showed how to use CLP analysis tools to analyse imperative programs. Their approach is based on having an interpreter, written in CLP, for the imperative language and then translating (small) imperative programs through partial evaluation of the interpreter.

Spoto et al. (2010) implement a termination analyzer for Java bytecode by expressing path-length reasoning as a CLP program and leveraging from existing CLP termination analysis tools. The resulting analyzer is robust and entirely automatic, covering the full language of Java bytecode. Albert et al. (2012) use a similar approach to cost analysis.

Morales et al. (2015) explore the use of a logic programming language for the implementation of efficient abstract machines and runtime systems. To this end they use a Prolog variant with certain imperative features (mutable variables) that enables translation into efficient C-style code while still allowing for high-level program transformations, such as partial evaluation of instruction definitions.

CLP has been also used as the basis for software model checking (Delzanno and Podelski 1999; Flanagan 2003) of concurrent systems and its use in software verification tools is rapidly growing. For example, it has been adopted in Threader (Gupta et al. 2011), UFO (Albarghouthi et al. 2012), SeaHorn (Gurfinkel et al. 2015), HSF (Grebenshchikov et al. 2012), VeriMAP (De Angelis et al. 2014), Eldarica (Rümmer et al. 2013), and TRACER (Jaffar et al. 2012). The task of encoding verification conditions is different to our aim of providing a platform for program compilation, although both require a convenient representation for reasoning about programs.

7 Conclusions

We have described Static Single Assignment form, and discussed a number of problems it causes for sophisticated analyses. Many of these problems have been previously addressed, but no previous work has addressed all of them. One approach that addressed several of these problems re-conceives a low-level program as a functional program.

We propose going further and viewing a low-level program as a logic program, and have suggested a simple, deterministic, strongly moded logic programming language as a compiler intermediate representation. The language is fully declarative; many existing analyses for logic programming languages will apply directly. We have presented a powerful analysis and transformation for this form. Because LP form uses procedure calls for all control transfer, operations that cross block boundaries are naturally inter-procedural. Owing to determinism and single-mode restrictions, LP form is surprisingly close to machine language, so final code generation is not difficult. Thus LP form is a suitable choice for a compiler's intermediate code representation.

We are currently developing an implementation of LP form, which we call LPVM. This is being used as intermediate representation for a compiler we are developing for a language combining the benefits of declarative and imperative programming. Since the procedures of the language support multiple outputs, that facility in LP form is particularly important. Rather than duplicating the extensive work of the LLVM project in producing high-quality, peep-hole optimised assembly language for multiple architectures, we plan to do all program analysis and transformation in LP form, and finally translate to LLVM for final code generation.

References

Albarghouthi, A., Li, Y., Gurfinkel, A., and Chechik, M. 2012. Ufo: A framework for abstraction- and interpolation-based software verification. In *Proc. 24rd Int. Conf. Computer Aided Verification*, P. Madhusudan and S. A. Seshia, Eds. Lecture Notes in Computer Science, vol. 7358. Springer, 672–678.

Albert, E., Arenas, P., Genaim, S., Puebla, G., and Zanardini, D. 2012. Cost analysis of object-oriented bytecode programs. *Theoretical Computer Science 413*, 142–159.

Alpern, B., Wegman, M. N., and Zadeck, F. K. 1988. Detecting equality of variables in programs. In *Proc. 15th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*. ACM, 1–11.

Ananian, C. S. 1999. The static single information form. M.S. thesis, Princeton University.

Appel, A. W. 1992. *Compiling with Continuations*. Cambridge University Press.

Appel, A. W. 1998. SSA is functional programming. *SIGPLAN Notices 33,* 4, 17–20.

Ballance, R. A., Maccabe, A. B., and Ottenstein, K. J. 1990. The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*. ACM, 257–271.

Benton, W. C. and Fischer, C. N. 2007. Interactive, scalable, declarative program analysis: From prototype to implementation. In *Proc. 9th ACM SIGPLAN Int. Conf. Principles and Practice of Declarative Programming*. ACM, 13–24.

Chow, F., Chan, S., Liu, S.-M., Lo, R., and Streich, M. 1996. Effective representation of aliases and indirect memory operations in SSA form. In *Compiler Construction*, T. Gyimóthy, Ed. Lecture Notes in Computer Science, vol. 1060. Springer, 253–267.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems 13,* 4, 451–490.

De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. 2014. VeriMAP: A tool for verifying programs through transformations. In *Proc. 20th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, A. Ábrahám and K. Havelund, Eds. Lecture Notes in Computer Science, vol. 8413. Springer, 568–574.

Delzanno, G. and Podelski, A. 1999. Model checking in CLP. In *Proc. 5th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, W. R. Cleaveland, Ed. Lecture Notes in Computer Science, vol. 1579. 223–239.

Flanagan, C. 2003. Automatic software model checking using CLP. In *Programming Languages and Systems: Proc. 12th European Symp. Programming*, P. Degano, Ed. Lecture Notes in Computer Science, vol. 2618. Springer, 189–203.

Gange, G., Navas, J., Schachte, P., Søndergaard, H., and Stuckey, P. J. 2015. Interval analysis and machine arithmetic: Why signedness ignorance is bliss. *ACM Transactions on Programming Languages and Systems 37,* 1, 1:1–1:35.

Gerlek, M. P., Stolz, E., and Wolfe, M. 1995. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Transactions on Programming Languages and Systems 17,* 1, 85–122.

Grebenshchikov, S., Lopes, N. P., Popeea, C., and Rybalchenko, A. 2012. Synthesizing software verifiers from proof rules. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*. ACM, 405–416.

Gupta, A., Popeea, C., and Rybalchenko, A. 2011. Threader: A constraint-based verifier for multi-threaded programs. In *Proc. 23rd Int. Conf. Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds. Lecture Notes in Computer Science, vol. 6806. Springer, 412–417.

Gurfinkel, A., Kahsai, T., and Navas, J. A. 2015. SeaHorn: A framework for verifying C programs (competition contribution). In *Proc. 21st Int. Conf. Tools and Algorithms for*

the Construction and Analysis of Systems, C. Baier and C. Timelli, Eds. Vol. 9035. Springer, 447–450.

Jaffar, J., Murali, V., Navas, J. A., and Santosa, A. E. 2012. TRACER: A symbolic execution tool for verification. In *Proc. 24th Int. Conf. Computer Aided verification*, P. Madhusudan and S. A. Seshia, Eds. Lecture Notes in Computer Science, vol. 7358. Springer, 758–766.

Kelsey, R. A. 1995. A correspondence between continuation passing style and static single assignment form. *SIGPLAN Notices 30,* 3, 13–22.

Lattner, C. and Adve, V. 2004. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proc. Int. Symp. Code Generation and Optimization (CGO'04)*. IEEE Comp. Soc., 75–86.

Miné, A. 2006. The octagon abstract domain. *Higher-Order and Symbolic Computation 19,* 1, 31–100.

Morales, J. F., Carro, M., and Hermenegildo, M. 2015. Description and optimization of abstract machines in a dialect of Prolog. *Theory and Practice of Logic Programming*. To appear.

Peralta, J. C. and Cruz-Carlón, J. A. 2006. From static single-assignment form to definite programs and back. In *Pre-Proceedings of 16th International Symposium on Logic Based Program Synthesis and Transformation (LOPSTR 2006)*, G. Puebla, Ed. 79–84.

Peralta, J. C., Gallagher, J. P., and Sağlam, H. 1998. Analysis of imperative programs through analysis of constraint logic programs. In *Static Analysis*, G. Levi, Ed. Lecture Notes in Computer Science, vol. 1503. Springer, 246–261.

Rümmer, P., Hojjat, H., and Kuncak, V. 2013. Disjunctive interpolants for Horn-clause verification. In *Proc. 25rd Int. Conf. Computer Aided Verification*, N. Sharygina and H. Veith, Eds. Lecture Notes in Computer Science, vol. 8044. 347–363.

Somogyi, Z., Henderson, F., and Conway, T. 1996. The execution algorithm of Mercury: An efficient purely declarative logic programming language. *Journal of Logic Programming 29,* 1–3, 17–64.

Spoto, F., Mesnard, R., and Payet, É. 2010. A termination analyzer for Java bytecode based on path-length. *ACM Transactions on Programming Languages and Systems 32*, 8:1–8:70.

Tu, P. and Padua, D. 1995. Efficient building and placing of gating functions. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*. ACM, 47–55.

Whaley, J., Avots, D., Carbin, M., and Lam, M. S. 2005. Using Datalog with binary decision diagrams for program analysis. In *Proc. Third Asian Symp. Programming Languages and Systems*, K. Yi, Ed. Lecture Notes in Computer Science, vol. 3780. Springer, 97–118.