Journal of the Egyptian Mathematical Society (2012) **20**, 28–37

Egyptian Mathematical Society

Journal of the Egyptian Mathematical Society

www.etms-eg.org
www.elsevier.com/locate/joems

ORIGINAL ARTICLE

Dead code elimination based pointer analysis for multithreaded programs

Mohamed A. El-Zawawy

Department of Mathematics, Faculty of Science, Cairo University, Giza 12316, Egypt

Available online 2 February 2012

Abstract This paper presents a new approach for optimizing multitheaded programs with pointer constructs. The approach has applications in the area of certified code (proof-carrying code) where a justification or a proof for the correctness of each optimization is required. The optimization meant here is that of dead code elimination.

Towards optimizing multithreaded programs the paper presents a new operational semantics for parallel constructs like join-fork constructs, parallel loops, and conditionally spawned threads. The paper also presents a novel type system for flow-sensitive pointer analysis of multithreaded programs. This type system is extended to obtain a new type system for live-variables analysis of multithreaded programs. The live-variables type system is extended to build the third novel type system, proposed in this paper, which carries the optimization of dead code elimination. The justification mentioned above takes the form of type derivation in our approach.

1. Introduction

One of the mainstream programming approaches today is multithreading. Using multiple threads is useful in many ways like (a) concealing suspension caused by some commands, (b) making it easier to build huge software systems, (c) improving execution of programs specially those that are executed on multiprocessors, and (d) building advanced user interfaces.

The potential interaction between threads in a multithreaded programs complicates both the compilation and the program analysis processes. Moreover this interaction also makes it difficult to extend the scope of program analysis techniques of sequential programs to cover multithreaded programs.

Typically optimizing multithreaded programs is achieved in an algorithmic form using data-flow analyses. This includes transforming the given program into a control-flow graph which is a convenient form for the algorithm to manipulate. For some applications like certified code, it is desirable to

E-mail address: maelzawawy@cu.edu.eg

Peer review under responsibility of Egyptian Mathematical Society.
doi:10.1016/j.joems.2011.12.011

Production and hosting by Elsevier

associate each program optimization with a justification or a proof for the correctness of the optimization. For these cases, the algorithmic approach to program analysis is not a good choice as it does not work on the syntactical structure of the program and hence does not reflect the transformation process. Moreover the desired justification must be relatively simple as it gets checked within trusted computing base.

Type systems stand as a convenient alternative for the algorithmic approach of program analyses when a justification is necessary. In the type systems approach, analysis and optimization of programs are directed by the syntactical structure of the program. Inference rules of type systems are advantageously relatively simple and so is the justification which takes the form of a type derivation in this case. The adequacy of type systems for program analysis has already been studied like in [3,12,22]

Pointer analysis is among the most important program analyses and it calculates information describing contents of pointers at different program points. The application of pointer analysis to multithreaded programs results in information that is required for program analyses and compiler optimizations such as live-variables analysis and dead code elimination, respectively. The live-variables analysis finds for each program point the set of variables whose values are used usefully in the rest of the program. The results of live-variables analysis is necessary for the optimization of dead code elimination which removes code that has no effect on values of variables of interest at the end of the program.

This paper presents a new approach for optimizing multithreaded programs with pointer constructs. The scope of the proposed approach is broad enough to include certified (proof-carrying) code applications where a justification for optimization is necessary. Type systems are basic tools of the new approach which considers structured parallel constructs like join-fork constructs, parallel loops, and conditionally spawned threads. The justifications in our approach take the form of type derivations. More precisely, the paper presents a type system for flow-sensitive pointer analysis of multithreaded programs. The live-variables analysis of multithreaded programs is also treated in this paper by a type system which is an extension of the type system for pointer analysis. The extension has the form of another component being added to points-to types. The dead code elimination of multithreaded programs is then achieved using a type system which is again an extension of the type system for live-variables analysis. This time the extension takes the form of a transformation component added to inference rules of the type system for live-variables analysis. To prove the soundness of the three proposed type systems, a novel operational semantics for parallel constructs is proposed in this paper.

1.1. Motivation

Fig. 1 presents a motivating example of the work presented in this paper. Consider the program on the left-hand-side of the figure. Suppose that at the end of the program we are interested in the values of x and y. We note that the assignment in line 8 is a dead code as the variable x is modified in line 9 before we make any use of the value that the variable gets in line 8. The assignment in line 2 indirectly modifies y which is modified again in the *par* command before any useful use of the value that y gets in line 2. Therefore line 2 is a dead code.

```
1.  x := &y;                     x := &y;
2.  *x := 2;                     skip;
3.  par{                         par{
4.      {y := 4}                     {y := 4}
5.      {y := 5;        ⟹           {skip;
6.       *x := 6}                     *x := 6}
7.      };                           };
8.  x := 8;                      skip;
9.  x := 9                       x := 9
```

Figure 1 A motivating example.

The *par* command has two threads which can be executed in any order. If the first thread is executed first then assignments in lines 4 and 5 become dead code. If the second thread is executed first then assignments in lines 5 and 6 become dead code. Therefore the dead code in the *par* command is the assignment in line 5 only.

This paper presents a technique that discovers and removes such dead code in parallel structured programs with pointer constructs. The output of the technique is a program like that on the right-hand-side of Fig. 1. In addition to the join-fork construct (*par*), the paper also considers other parallel constructs like conditionally spawned threads and parallel loops. With each such program optimization, our technique presents a justification or a proof for the correctness of the optimization. The proof takes the form of a type derivation.

1.2. Contributions

Contributions of this paper are the following:

1. A simple yet powerful operational semantics for multithreaded programs with pointer constructs.
2. A novel type system for pointer analysis of multithreaded programs. To our knowledge, this is the first attempt to use type systems for pointer analysis of multithreaded programs.
3. A new type systems for live-variables analysis of multithreaded programs.
4. An original type system for the optimization of dead code elimination for multithreaded programs.

1.3. Organization

The rest of the paper is organized as follows. The language that we study (the while language enriched with pointer and parallel constructs) and an operational semantics for its constructs are presented in Section 2. Sections 3 and 4 present our proposed type systems for flow-sensitive pointer and live-variables analyses, respectively. The type system carrying program optimization is introduced in Section 5. Related work is discussed in Section 6.

2. Programming language

This section presents the programming language (Fig. 2) we use together with an operational semantics for its constructs. The language is the simple *while* language [8] enriched with commands for pointer manipulations and structured parallel constructs.

$$\begin{aligned}
& n \in \mathbb{Z},\ x \in Var,\ and \oplus \in \{+,-,\times\} \\
e \in Aexprs &::= x \mid n \mid e_1 \oplus e_2 \\
b \in Bexprs &::= true \mid false \mid \neg b \mid e_1 = e_2 \mid e_1 \le e_2 \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \\
S \in Stmts &::= x := e \mid x := \&y \mid *x := e \mid x := *y \mid skip \mid S_1; S_2 \mid if\ b\ then\ S_t\ else\ S_f \mid \\
& \quad while\ b\ do\ S_t \mid par\{\{S_1\},\ldots,\{S_n\}\} \mid par\text{-}if\{(b_1,S_1),\ldots,(b_n,S_n)\} \mid par\text{-}for\{S\}.
\end{aligned}$$

Figure 2 The programming language.

The parallel constructs include join-fork constructs, parallel loops, and conditionally spawned threads. The *par* (join-fork) construct starts executing many concurrent threads at the beginning of the *par* construct and then waits until the completion of all these executions at the end of the *par* construct. Semantically, the *par* construct can be expressed approximately as if the threads are executed sequentially in an arbitrary order. The parallel loop construct included in our language is that of *par-for*. This construct executes, in parallel, a statically unknown number of threads each of which has the same code (the loop body). Therefore the semantics of *par-for* can be expressed using that of the *par* construct. The construct including conditionally spawned threads is that of *par-if*. This construct executes, in parallel, its n concurrent threads. The execution of thread (b_i, S_i) includes the execution of S_i only if b_i is true.

One way to define the meaning of the constructs of our programming language, including the parallel constructs, is by an operational semantics. This amounts to defining a transition relation $\rightsquigarrow$ between states which are defined as follows.

Definition 1

1. $Addrs = \{x' \mid x \in Var\}$ and $Val = \mathbb{Z} \cup Addrs$.
2. *A state is either an abort or a map* $\gamma \in \Gamma = Var \to Val$.

The semantics of arithmetic and Boolean expressions are defined as usual except that arithmetic and Boolean operations are not allowed on pointers.

$$[\![n]\!]\gamma = n \quad [\![\&x]\!]\gamma = x' \quad [\![x]\!]\gamma = \gamma(x) \quad [\![true]\!]\gamma = true$$

$$[\![false]\!]\gamma = false$$

$$[\![*x]\!]\gamma = \begin{cases} \gamma(y) & \text{if } \gamma(x) = y', \\ ! & \text{otherwise.} \end{cases}$$

$$[\![e_1 \oplus e_2]\!]\gamma = \begin{cases} [\![e_1]\!]\gamma \oplus [\![e_2]\!]\gamma & \text{if } [\![e_1]\!]\gamma, [\![e_2]\!]\gamma \in \mathbb{Z}, \\ ! & \text{otherwise.} \end{cases}$$

$$[\![\neg A]\!]\gamma = \begin{cases} \neg([\![A]\!]\gamma) & \text{if} [\![A]\!]\gamma \in \{true, false\}, \\ ! & \text{otherwise.} \end{cases}$$

$$[\![e_1 = e_2]\!]\gamma = \begin{cases} ! & \text{if } [\![e_1]\!]\gamma = ! \text{ or } [\![e_2]\!]\gamma = !, \\ true & \text{if } [\![e_1]\!]\gamma = [\![e_2]\!]\gamma \neq !, \\ false & \text{otherwise.} \end{cases}$$

$$[\![e_1 \leqslant e_2]\!]\gamma = \begin{cases} ! & \text{if } [\![e_1]\!]\gamma \notin \mathbb{Z} \text{ or } [\![e_2]\!]\gamma \notin \mathbb{Z}, \\ [\![e_1]\!]\gamma \leqslant [\![e_2]\!]\gamma & \text{otherwise.} \end{cases}$$

For $\diamondsuit \in \{\wedge, \vee\}$, $[\![b_1 \diamondsuit b_2]\!]\gamma$

$$= \begin{cases} ! & \text{if } [\![b_1]\!]\gamma = ! \text{or} [\![b_2]\!]\gamma = !, \\ [\![b_1]\!]\gamma \diamondsuit [\![b_2]\!]\gamma & \text{otherwise.} \end{cases}$$

The inference rules of our semantics (transition relation) are defined as follows:

$$\frac{[\![e]\!]\gamma = !}{x := e : \gamma \rightsquigarrow abort} \qquad \frac{[\![e]\!]\gamma \neq !}{x := e : \gamma \rightsquigarrow \gamma[x \mapsto [\![e]\!]\gamma]}$$

$$\frac{\gamma(x) = z' \quad z := e : \gamma \rightsquigarrow state}{*x := e : \gamma \rightsquigarrow state}$$

$$\frac{\gamma(x) \notin Addrs}{*x := e : \gamma \rightsquigarrow abort} \qquad \frac{}{x := \&y : \gamma \rightsquigarrow \gamma[x \mapsto y']}$$

$$\frac{\gamma(y) = z' \quad x := z : \gamma \rightsquigarrow \gamma'}{x := *y : \gamma \rightsquigarrow \gamma'}$$

$$\frac{\gamma(y) \notin Addrs}{x := *y : \gamma \rightsquigarrow abort} \qquad \frac{}{skip : \gamma \rightsquigarrow \gamma} \qquad \frac{S_1 : \gamma \rightsquigarrow abort}{S_1; S_2 : \gamma \rightsquigarrow abort}$$

$$\frac{S_1 : \gamma \rightsquigarrow \gamma'' \quad S_2 : \gamma'' \rightsquigarrow state}{S_1; S_2 : \gamma \rightsquigarrow state}$$

$$\frac{[\![b]\!]\gamma = !}{if\ b\ then\ S_t\ else\ S_f : \gamma \rightsquigarrow abort} \qquad \frac{[\![b]\!]\gamma = true \quad S_t : \gamma \rightsquigarrow state}{if\ b\ then\ S_t\ else\ S_f : \gamma \rightsquigarrow state}$$

$$\frac{[\![b]\!]\gamma = false \quad S_f : \gamma \rightsquigarrow state}{if\ b\ then\ S_t\ else\ S_f : \gamma \rightsquigarrow state} \qquad \frac{[\![b]\!]\gamma = !}{while\ b\ do\ S_t : \gamma \rightsquigarrow abort}$$

$$\frac{[\![b]\!]\gamma = false}{while\ b\ do\ S_t : \gamma \rightsquigarrow \gamma}$$

$$\frac{[\![b]\!]\gamma = true \quad S : \gamma \rightsquigarrow \gamma'' \quad while\ b\ do\ S_t : \gamma'' \rightsquigarrow state}{while\ b\ do\ S_t : \gamma \rightsquigarrow state}$$

$$\frac{[\![b]\!]\gamma = true \quad S : \gamma \rightsquigarrow abort}{while\ b\ do\ S_t : \gamma \rightsquigarrow abort}$$

- *Join-fork:*

$$\frac{}{par\{\{S_1\},\ldots,\{S_n\}\} : \gamma \rightsquigarrow \gamma'}\dagger \qquad \frac{}{par\{\{S_1\},\ldots,\{S_n\}\} : \gamma \rightsquigarrow abort}\ddagger$$

† there exist a permutation θ: $\{1,\ldots,n\} \to \{1,\ldots,n\}$ and $n+1$ states $\gamma = \gamma_1,\ldots,\gamma_{n+1} = \gamma'$ such that for every $1 \leqslant i \leqslant n$, $S_{\theta(i)}$: $\gamma_i \to \gamma_{i+1}$.

‡ there exist m such that $1 \leqslant m \leqslant n$, a one-to-one map β: $\{1,\ldots,m\} \to \{1,\ldots,n\}$, and $m+1$ states $\gamma = \gamma_1$, $\ldots$, $\gamma_{m+1} = abort$ such that for every $1 \leqslant i \leqslant m$, $S_{\beta(i)}$: $\gamma_i \to \gamma_{i+1}$.

- *Conditionally spawned threads:*

$$\frac{par\{\{if\, b_1\, then\, S_1\, else\, skip\},\ldots,\{if\, b_n\, then\, S_n\, else\, skip\}\}:\gamma\rightsquigarrow\gamma'}{par-if\{(b_1,S_1),\ldots,(b_n,S_n)\}:\gamma\rightsquigarrow\gamma'}$$

$$\frac{par\{\{if\, b_1\, then\, S_1\, else\, skip\},\ldots,\{if\, b_n\, then\, S_n\, else\, skip\}\}:\gamma\rightsquigarrow abort}{par-if\{(b_1,S_1),\ldots,(b_n,S_n)\}:\gamma\rightsquigarrow abort}$$

- *Parallel loops:*

$$\frac{\exists n.\, par\,\overbrace{\{\{S\},\ldots,\{S\}\}}^{n-times}:\gamma\rightsquigarrow\gamma'}{par-for\,\{S\}:\gamma\rightsquigarrow\gamma'} \qquad \frac{\exists n.\, par\,\overbrace{\{\{S\},\ldots,\{S\}\}}^{n-times}:\gamma\rightsquigarrow abort}{par-for\{S\}:\gamma\rightsquigarrow abort}$$

3. Pointer analysis

In this section, we present a novel technique for flow-sensitive pointer analysis of structured parallel programs where shared pointers may be updated simultaneously. Our technique manipulates important parallel constructs; join-fork constructs, parallel loops, and conditionally spawned threads. The proposed technique has the form of a compositional type system which is simply structured. Consequently results of the analysis are in the form of types assigned to expressions and statements approved by type derivations. Therefore a type is assigned to each program point of a statement (program). This assigned type specifies for each variable in the program a conservative approximation of the addresses that may get into the variable. The set of points-to types PTS and the relation $\models\subseteq\Gamma\times PTS$ are defined as follows:

Definition 2

1. $PTS = \{pts \mid pts: Var \to 2^{Addrs}\}$.
2. $pts \leqslant pts' \overset{def}{\Longleftrightarrow} \forall x \in Var \cdot pts(x) \subseteq pts'(x)$.
3. $\gamma \models pts \overset{def}{\Longleftrightarrow} (\forall x \in Var \cdot \gamma(x) \in Addrs \Rightarrow \gamma(x) \in pts(x))$.

The inference rules of our type system for pointer analysis are the following:

$$\frac{}{n:pts\to\emptyset} \qquad \frac{}{x:pts\to pts(x)} \qquad \frac{}{e_1\oplus e_2:pts\to\emptyset}$$

$$\frac{e:pts\to A}{x:=e:pts\to pts[x\mapsto A]}(:=^p)$$

$$\frac{}{x:=\&y:pts\to pts[x\mapsto\{y'\}]}(:=\&^p) \qquad \frac{}{skip:pts\to pts}$$

$$\frac{\forall z'\in pts(y).\ x:=z:pts\to pts'}{x:=*y:pts\to pts'}(:=*^p)$$

$$\frac{\forall z'\in pts(x).\ z:=e:pts\to pts'}{*x:=e:pts\to pts'}(*:=^p)$$

$$\frac{S_i:pts\cup\cup_{j\neq i}pts_j\to pts_i}{par\{\{S_1\},\ldots,\{S_n\}\}:pts\to\cup_i pts_i}(par^p)$$

$$\frac{S_1:pts\to pts''\quad S_2:pts''\to pts'}{S_1;S_2:pts\to pts'}(seq^p)$$

$$\frac{par\{\{if\, b_1\, then\, S_1\, else\, skip\},\ldots,\{if\, b_n\, then\, S_n\, else\, skip\}\}:pts\to pts'}{par-if\{(b_1,S_1),\ldots,(b_n,S_n)\}:pts\to pts'}(par-if^p)$$

$$\frac{S:pts\cup pts'\to pts'}{par-for\{S\}:pts\to pts'}(par-for^p)$$

$$\frac{S_t:pts\to pts'\quad S_f:pts\to pts'}{if\, b\, then\, S_t\, else\, S_f:pts\to pts'}(if^p)$$

$$\frac{S_t:pts\to pts}{while\, b\, do\, S_t:pts\to pts}(whl^p)$$

$$\frac{pts_1'\leqslant pts_1\quad S:pts_1\to pts_2\quad pts_2\leqslant pts_2'}{S:pts_1'\to pts_2'}(csq^p)$$

The judgement of an expression has the form e: $pts \to A$. The intended meaning of this judgment, which is formalized in Lemma 1, is that A is the collection of addresses that e may evaluate to in a state of type pts. The judgement of a statement has the form S: $pts \to pts'$. This judgement simply guarantees that if S is executed in a state of type pts and the execution terminates in a state γ', then γ' has type pts'. Typically, the pointer analysis for a program S is achieved via a post-type derivation for the bottom type (mapping variables to $\emptyset$) as the pre-type.

The inference rules corresponding to assignment commands are clear. For the rule (par^p) of the join-fork command, *par*, one possibility is that the execution of a thread S_i starts before the execution of any other thread starts. Another possibility is that the execution starts after executions of all other threads end. Of course there are many other possibilities in between. Consequently, the analysis of the thread S_i must consider all such possibilities. This is reflected in the pre-type of S_i and the post-type of the *par* command. Similar explanations clarify the rules ($par - if^p$) and ($par - for^p$).

We note that a type invariant is required to type a *while* statement. Also to achieve the analysis for one of the *par*'s threads we need to know the analysis results for all other threads. However obtaining these results requires the result of analyzing the first thread. Therefore there is a kind of circularity in rule (par^p). Similar situations are in rules ($par - if^p$) and ($par - for^p$). Such issues can be treated using a fix-point algorithm. The convergence of this algorithm is guaranteed as the rules of our type system are monotone and the set of points-to types PTS is a complete lattice.

Lemma 1

1. *Suppose* e: $pts \to A$ *and* $\gamma \models pts$. *Then* $[\![e]\!]\gamma \in Addrs$ *implies* $[\![e]\!]\gamma \in A$.
2. $pts \leqslant pts' \iff (\forall\gamma.\ \gamma\models pts \Rightarrow \gamma\models pts')$.

Proof. The first item is obvious. The left-to-right direction of (2) is easy. The other direction is proved as follows. Suppose $y' \in pts\,(x)$. Then the state $\{(x,y'),(t,0) \mid t \in Var\backslash\{x\}\}$ is of type pts and hence of type pts' implying that $y' \in pts'(x)$. Therefore $pts(x) \subseteq pts'(x)$. Since x is arbitrary, $pts \leqslant pts'$. □

Theorem 1. *(Soundness) Suppose that* S: $pts \to pts'$, S: $\gamma \rightsquigarrow \gamma'$, *and* $\gamma \models pts$. *Then* $\gamma' \models pts'$.

Proof. The proof is by structure induction on the type derivation. We demonstrate some cases.

– The case of ($:=^p$): In this case $pts' = pts[x \mapsto A]$ and $\gamma' = \gamma[x \mapsto [\![e]\!]\gamma]$. Therefore by the previous lemma $\gamma \models pts$ implies $\gamma' \models pts'$.

– The case of $(*:=^p)$: In this case there exists $z \in Var$ such that $\gamma(x) = z'$ and $z := e$: $\gamma \rightsquigarrow \gamma'$. Because $\gamma \vDash pts$, $z' \in pts(x)$ and hence by assumption $z := e$: $pts \;\; \rightarrow pts'$. Therefore by soundness of $(:=^p)$, $\gamma' \vDash pts'$.
– The case of (par^p): In this case there exist a permutation θ: $\{1,\ldots,n\} \rightarrow \{1,\ldots,n\}$ and $n+1$ states $\gamma = \gamma_1, \ldots, \gamma_{n+1} = \gamma'$ such that for every $1 \leqslant i \leqslant n$, $S_{\theta(i)}$: $\gamma_i \rightarrow \gamma_{i+1}$. Also $\gamma_1 \vDash pts$ implies $\gamma_1 \vDash pts \cup \cup_{j \neq \theta(1)} pts_j$. Therefore by the induction hypothesis $\gamma_2 \vDash \;\; pts_{\theta(1)}$. This implies $\gamma_2 \vDash pts \cup \cup_{j \neq \theta(2)} pts_j$. Again by the induction hypothesis we get $\gamma_3 \vDash pts_{\theta(2)}$. Therefore by a simple induction on n, we can show that $\gamma' = \gamma_{n+1} \vDash \;\; pts_{\theta(n)}$ which implies $\gamma' \vDash \;\; pts' = \cup_j pts_j$.
– The case of $(par-for^p)$: In this case there exists n such that $par\,\{\overbrace{\{S\},\ldots,\{S\}}^{n-times}\} : \gamma \rightsquigarrow \gamma'$. By induction hypothesis we have S: $pts \cup \;\; pts' \rightarrow pts'$. By (par^p) we conclude that $par\,\{\overbrace{\{S\},\ldots,\{S\}}^{n-times}\} : pts \rightsquigarrow pts'$. Therefore by the soundness of (par^p), $\gamma' \vDash pts'$. $\square$

4. Live-variables analysis

In this section, we present a type system to perform live-variables analysis for pointer programs with structured parallel constructs. We start with defining live-variables:

Definition 3

– A variable is *usefully used* if it is used
  - as the operand of the unary operation $*$.
  - in an assignment to a variable that is live at the end of the assignment, or
  - in the guard of an if-statement or a while-statement,
– A variable is *live* at a program point if there is a computational path from that program point during which the variable gets usefully used before being modified.

Definition 4. The set of *live types* is denoted by L and equal to $pts \times \mathscr{P}(Var)$. The second component of a live type is termed a *live-component*. The subtyping relation $\leqslant$ is defined as: $(pts,l) \leqslant (pts',l') \overset{def}{\Longleftrightarrow} pts \leqslant pts'$ and $l \supseteq l'$.

The live-variables analysis is a backward analysis. For each program point, this analysis specifies the set of variables that may be live (according to the definition above) at that point.

Our type system for live-variables analysis is obtained as an enrichment of the type system for pointer analysis, presented in the previous section. Hence one can say that the type system presented here is a strict extension of that presented above. This is so because the result of pointer analysis is necessary to improve the precision of the live-variables analysis. This also gives an intuitive explanation of the definition of live types above.

The judgement of a statement S has the form $S{:}(pts,l) \rightarrow (pts',l')$. The intuition of the judgement is that the presence of live-variables at the post-state of an execution of S in l' implies the presence of live-variables at the pre-state of this execution in l. The intuition agrees with the fact that live-variables analysis is a backward analysis and gives an insight into the definition of $\gamma \vDash l$ below.

Suppose we have the set of variables l' that we have interest in their values at the end of executing a statement S and the result of pointer analysis of S (in the form S: $pts \;\; \rightarrow pts'$). The live-variables analysis takes the form of a pre-type derivation that calculates a set l such that $S{:}(pts\;,l) \rightarrow (pts',l')$.

The inference rules for our type system for live-variables analysis are as follows.

$$\frac{x := e : pts \;\rightarrow\; pts' \quad x \notin l'}{x := e : (pts, l') \;\rightarrow\; (pts', l')}\,(:=^l_1)$$

$$\frac{x := e : pts \;\rightarrow\; pts' \quad x \in l'}{x := e : (pts, (l' \setminus \{x\}) \cup FV(e)) \;\rightarrow\; (pts', l')}\,(:=^l_2)$$

$$\frac{}{x := \&y : (pts, l' \setminus \{x\}) \;\rightarrow\; (pts[x \mapsto \{y'\}], l')}\,(:= \&^l)$$

$$\frac{}{skip : (pts, l) \;\rightarrow\; (pts, l)}$$

$$\frac{x := *y : pts \;\rightarrow\; pts' \quad x \notin l'}{x := *y : (pts, l' \cup \{y\}) \;\rightarrow\; (pts', l')}\,(:= *^l_1)$$

$$\frac{x := *y : pts \;\rightarrow\; pts' \quad x \in l'}{x := *y : (pts, (l' \setminus \{x\}) \cup \{y, z | z' \in pts(y)\}) \;\rightarrow\; (pts', l')}\,(:= *^l_2)$$

$$\frac{*x := e : pts \;\rightarrow\; pts' \quad pts(x) \cap l' = \emptyset}{*x := e : (pts, l' \cup \{x\}) \;\rightarrow\; (pts', l')}\,(* :=^l_1)$$

$$\frac{*x := e : pts \;\rightarrow\; pts' \quad pts(x) \cap l' \neq \emptyset}{*x := e : (pts, l' \cup FV(e) \cup \{x\}) \;\rightarrow\; (pts', l')}\,(* :=^l_2)$$

$$\frac{S_i : (pts \cup \cup_{j \neq i} pts_j, l_i) \;\rightarrow\; (pts_i, l' \cup \cup_{j \neq i} l_j)}{par\{\{S_1\}, \ldots, \{S_n\}\} : (pts, \cup_i l_i) \;\rightarrow\; (\cup_i pts_i, l')}\,(par^l)$$

$$\frac{par\{\{if\, b_1\; then\; S_1\; else\; skip\}, \ldots, \{if\, b_n\; then\; S_n\; else\; skip\}\} : (pts,l) \rightarrow (pts',l')}{par-if\{(b_1,S_1),\ldots,(b_n,S_n)\} : (pts,l) \rightarrow (pts',l')}\,(par-if^l)$$

$$\frac{S : (pts \cup pts', l) \rightarrow (pts', l' \cup l)}{par-for\{S\} : (pts,l) \rightarrow (pts',l')}\,(par-for^l)$$

$$\frac{S_1 : (pts,l) \rightarrow (pts'',l'') \quad S_2 : (pts'',l'') \rightarrow (pts',l')}{S_1;S_2 : (pts,l) \rightarrow (pts',l')}\,(seq^l)$$

$$\frac{S_t, S_f : (pts,l) \rightarrow (pts',l')}{if\, b\; then\; S_t\; else\; S_f : (pts, l \cup FV(b)) \rightarrow (pts',l')}\,(if^l)$$

$$\frac{l = l' \cup FV(b) \quad S_t : (pts,l') \rightarrow (pts,l)}{while\; b\; do\; S_t : (pts,l) \rightarrow (pts,l')}\,(whl^l)$$

$$\frac{(pts'_1,l'_1) \leqslant (pts_1,l_1) \quad S : (pts_1,l_1) \rightarrow (pts_2,l_2) \quad (pts_2,l_2) \leqslant (pts'_2,l'_2)}{S : (pts'_1,l'_1) \rightarrow (pts'_2,l'_2)}\,(csq^l)$$

For the command $*x := e$, we have two rules, namely $\left(* :=^l_1\right)$ and $\left(* :=^l_2\right)$. In both cases, calculating the pre-type from the post-type includes adding x to the post-type. This is so because according to Definition 3, x is live at the pre-state of any execution of the command. The rule $\left(* :=^l_1\right)$ deals with the case that there is no possibility that the modified variable by this statement is live ($pts\;(x) \cap l' = \emptyset$) at the end of an execution. In this case there is no need to add any other variables to the post-type. The rule $\left(* :=^l_2\right)$ deals with the case that there is a possibility that the modified variable by this statement is live ($pts\;(x) \cap l' \neq \emptyset$) at the end of an execution. In this case, there is a possibility that free variables of e are used usefully according to Definition 3. Therefore free variables of e are added to the post-type. This gives an intuitive explanation for rules of all the assignment commands. The intuition given in the previous section for the rules (par^p) helps to understand the rules for the parallel constructs, (par^l), ($par - if^l$), and ($par - for^l$).

Towards proving the soundness of our type system for live-variables analysis, we introduce necessary definitions and results.

Definition 5

1. $\gamma \models_l pts \overset{\text{def}}{\iff} \forall x \in l.\ \gamma(x) \in Addrs \Rightarrow \gamma(x) \in pts(x)$.
2. $\gamma \sim_l \gamma' \overset{\text{def}}{\iff} \forall x \in l.\ \gamma(x) = \gamma'(x)$.
3. $\gamma \sim_{(pts,l)} \gamma' \overset{\text{def}}{\iff} \gamma \models_l pts, \gamma' \models_l pts$, *and* $\gamma \sim_l \gamma'$.

Definition 6. The expression $\gamma \vDash l$ denotes the case when there is a variable that is live at that state (computational point) and is not included in l. A state γ has type (pts,l), denoted by $\gamma \vDash (pts,l)$, if $\gamma \vDash_l pts$ and $\gamma \vDash l$.

The following lemma is proved by structure induction on e and b.

Lemma 2. *Suppose that* γ *and* γ' *are states and* l and $l' \in \mathscr{P}(Var)$. *Then*

1. *If* $l \supseteq l'$ *and* $\gamma \sim_l \gamma'$, *then* $\gamma \sim_{l'} \gamma'$.
2. *If* $l = l' \cup FV(e)$ *and* $\gamma \sim_l \gamma'$, *then* $[\![e]\!]\gamma = [\![e]\!]\gamma'$ *and* $\gamma \sim_{l'} \gamma'$.
3. *If* $l = l' \cup FV(b)$ *and* $\gamma \sim_l \gamma'$, *then* $[\![b]\!]\gamma = [\![b]\!]\gamma'$ *and* $\gamma \sim_{l'} \gamma'$.

The following lemma follows from Lemma 1.

Lemma 3. *Suppose that* $\gamma \vDash_l pts, FV(e) \subseteq l$, *and* $e: pts \to A$. *Then*

$[\![e]\!]\gamma \in Addrs \ \rightarrow\ [\![e]\!]\gamma \in A$.

Proof. Consider the state γ', where $\gamma' = \lambda\, x.$ if $x \in FV(e)$ then $\gamma(x)$ else 0. It is not hard to see that $[\![e]\!]\gamma = [\![e]\!]\gamma'$ and $\gamma' \vDash pts$. Now by Lemma 1, $[\![e]\!]\gamma' \in Addrs$ implies $[\![e]\!]\gamma' \in A$ which completes the proof. □

Theorem 2

1. $(pts, l) \leqslant (pts', l') \Rightarrow (\forall \gamma.\ \gamma \models_l pts \rightarrow \gamma \models_{l'} pts')$.
2. *Suppose that* $S{:}(pts,l) \to (pts',l')$ *and* $S{:}\ \gamma \rightsquigarrow \gamma'$. *Then* $\gamma \vDash_l pts$ *implies* $\gamma' \models_{l'} pts'$.
3. *Suppose that* $S{:}(pts,l) \to (pts',l')$ *and* $S{:}\ \gamma \rightsquigarrow \gamma'$. *Then* $\gamma \vDash l$ *implies* $\gamma' \vDash l'$. *This guarantees that if the set of variables live at* γ' *is included in* l', *then the set of variables live at* γ *is included in* l.

Proof

1. Suppose $\gamma \vDash_l pts$. This implies $\gamma \models_{l'} pts$ because $l' \subseteq l$. The last fact implies $\gamma \models_{l'} pts'$ because $pts \leqslant pts'$.
2. The proof is by induction on the structure of type derivation. We show some cases.
   (a) The type derivation has the form $(:=_1^l)$. In this case, $pts' = pts\ [x \mapsto A]$ and $\gamma' = \gamma[x \mapsto [\![e]\!]\ \gamma]$. Therefore $\gamma \models_{l'} pts$ implies $\gamma' \models_{l'} pts'$ because $x \notin l'$.
   (b) The type derivation has the form $(:=_2^l)$. In this case, $e{:}\ pts \to A$, $pts' = pts[x \mapsto A]$, $\gamma' = \gamma[x \mapsto [\![e]\!]\gamma]$, and $l = (l' \backslash \{x\}) \cup FV(e)$. Therefore by Lemma 3 it is not hard to see $\gamma' \models_{l'} pts'$.
   (c) The type derivation has the form $(:= *_1^l)$. In this case, for every $z' \in pts(y)$, we have $x{:=}z{:}\ pts \to pts'$, $\gamma(y) = z'$, and $x{:=}z{:}\ \gamma \to \gamma'$. We have $z' \in pts(y)$, because $y \in l$ and $\gamma \vDash_l pts$. Therefore by $(:=_1^l)$, we have $x{:=}z{:}(pts\ ,l') \to (pts',l')$. Now $\gamma \vDash_l pts$ amounts to $\gamma \models_{l'} pts$. Hence we get $\gamma' \models_{l'} pts'$ by soundness of $(:=_1^l)$.
   (d) The type derivation has the form $(:= *_2^l)$. In this case, for every $z' \in pts(y)$, we have $x{:=}z{:}\ pts \to pts'$, $\gamma(y) = z'$, $x{:=}z{:}\ \gamma \to \gamma'$, and $l = (l' \backslash \{x\}) \cup \{y, z | z' \in pts(y)\}$. We have $z \in pts(y)$ because $\gamma \vDash_l pts$ and $y \in l$. Therefore by $(:=_2^l)$ we have $x{:=}\ z{:}(pts, (l' \backslash \{x\}) \cup \{z\}) \to (pts',\ l')$. $\gamma \vDash_l pts$ implies $\gamma \models_{(l' \backslash \{x\}) \cup \{z\}} pts$. Hence by soundness of $(:=_2^l)$, we get $\gamma' \models_{l'} pts'$.
   (e) The type derivation has the form $(* :=_1^l)$. In this case, for every $z' \in pts(x)$, we have $z{:=}e{:}\ pts \to pts'$, $\gamma(x) = z'$, and $z{:=}e{:}\ \gamma \to \gamma'$. We have $z' \in pts(x)$, because $x \in l$ and $\gamma \vDash_l pts$. Therefore by $(:=_1^l)$, we have $z{:=}e{:}(pts\ ,l') \to (pts',l')$ because $pts(x) \cap l' = \emptyset$. Now $\gamma \vDash_l pts$ amounts to $\gamma \models_{l'} pts$. Hence we get $\gamma' \models_{l'} pts'$ because $z{:=}\ e{:}(pts,l') \to (pts',l')$ and by soundness of $(:=_1^l)$.
   (f) The type derivation has the form $(* :=_2^l)$. In this case, for every $z' \in pts(x)$, we have $z{:=}e{:}\ pts \to pts'$, $\gamma(x) = z'$, $z{:=}e{:}\ \gamma \to \gamma'$, and $l = l' \cup FV(e) \cup \{x\}$. We have $z \in pts(x)$ because $\gamma \vDash_l pts$ and $x \in l$. Therefore by $(:=_2^l)$ we have $x{:=}z{:}(pts, (l' \backslash \{z\}) \cup FV(e)) \to (pts',l')$. $\gamma \vDash_l pts$ implies $\gamma \models_{(l' \backslash \{z\}) \cup FV(e)} pts$. Hence by soundness of $(:=_2^l)$, we get $\gamma' \models_l' pts'$.
   (g) The type derivation has the form (par^l). In this case there exist a permutation θ: $\{1, \ldots, n\} \to \{1, \ldots, n\}$ and $n + 1$ states $\gamma = \gamma_1, \ldots, \gamma_{n+1} = \gamma'$ such that for every $1 \leqslant i \leqslant n$, $S_{\theta(i)}{:}\ \gamma_i \to \gamma_{i+1}$. Also $\gamma_1 \vDash_l pts$ implies $\gamma_1 \models_{l_{\theta(1)}} pts \cup \cup_{j \neq \theta(1)} pts_j$. Therefore by the induction hypothesis $\gamma_2 \models_{l' \cup \cup_{j \neq \theta(1)} l_j} pts_{\theta(1)}$. This implies $\gamma_2 \models_{l_{\theta(2)}} pts \cup \cup_{j \neq \theta(2)} pts_j$. Again by the induction hypothesis we get $\gamma_3 \models_{l' \cup \cup_{j \neq \theta(2)} l_j} pts_{\theta(2)}$. Therefore by a simple induction on n, we can show that $\gamma' = \gamma_{n+1} \models_{l' \cup \cup_{j \neq \theta(n)} l_j} pts_{\theta(n)}$ which implies $\gamma' \models_{l'} pts' = \cup_j pts_j$.
   (h) The type derivation has the form ($par - for^l$): In this case there exists n such that $par\{\overbrace{\{S\}, \ldots, \{S\}}^{n-times}\}$: $\gamma \rightsquigarrow \gamma'$. By induction hypothesis we have S: $(pts \cup pts',l) \to (pts',l \cup l')$. By ($par^l$) we conclude that $par\{\overbrace{\{S\}, \ldots, \{S\}}^{n-times}\} : (pts, l) \rightarrow (pts', l')$. Therefore by soundness of (par^l), we get $\gamma' \models_{l'} pts'$.
3. The proof is also by induction on the structure of type derivation and it is straightforward. □

Algorithm: *parallel-optimize*

- Input : a statement S of the language presented in Section 2 and a set of variables l' that we consider live (their values concern us) at the end of executing S;
- Output: an optimized and may be corrected version S' of S such that the relation between S and S' is as stated in Theorem 4.
- Method :
  1. Find pts such that $S : \bot \rightarrow pts$ in the type system for pointer analysis.
  2. Find l such that $S : (\bot, l) \rightarrow (pts, l')$ in the type system for live-variables analysis.
  3. Find S' such that $S : (\bot, l) \rightarrow (pts, l') \hookrightarrow S'$ in the type system for dead code elimination.

Figure 3 The algorithm *optimize-parallel*.

The proof of the following corollary follows from Theorem 2.

Corollary 1. *Suppose* $S: \gamma \rightsquigarrow \gamma'$ *and* $S: (pts,l) \rightarrow (pts',l')$. *Then* $\gamma \models (pts,l)$ *implies* $\gamma' \models (pts',l')$.

Theorem 3. *Suppose that* $S: (pts,l) \rightarrow (pts',l')$, $S: \gamma \rightsquigarrow \gamma'$, $\gamma \sim_{(pts,l)} \gamma_*$, *and S does not abort at* γ_*. *Then there exists a state* γ'_* *such that* $S : \gamma_* \rightarrow \gamma'_*$ *and* $\gamma' \sim_{(pts',l')} \gamma'_*$.

Proof. The proof is by induction on structure of type derivation. We demonstrate some cases:

1. The type derivation has one of the forms $(:=^l_1)$ and $(:=^l_2)$. In this case, $pts' = pts[x \mapsto A]$ and $\gamma' = \gamma[x \mapsto [\![e]\!]\gamma]$. We take $\gamma'_* = \gamma_*[x \mapsto [\![e]\!]\gamma_*]$.
2. The type derivation has the form $(:= *^l_1)$ or $(:= *^l_2)$. In this case, $\forall z' \in pts(y)$, we have $x{:=}z$: $pts \rightarrow pts'$,$\gamma(y) = z'$, and $x{:=}\ z$: $\gamma \rightarrow \gamma'$. We set $\gamma'_* = \gamma_*[x \mapsto \gamma_*(z)]$.
3. The type derivation has one of the forms $(:= *^l_1)$ and $(:= *^l_2)$. In this case, $\forall z' \in pts(x)$, we have $z{:=}e$: $pts \rightarrow pts'$,$\gamma(x) = z'$, and $z{:=}\ e$: $\gamma \rightarrow \gamma'$. We let $\gamma'_* = \gamma_*[z \mapsto [\![e]\!]\gamma_*]$
4. The type derivation has the form (par^l). In this case there exist a permutation θ: $\{1, \ldots ,n\} \rightarrow \{1, \ldots ,n\}$ and $n + 1$ states $\gamma = \gamma_1, \ldots ,\gamma_{n+1} = \gamma'$ such that for every $1 \leqslant i \leqslant n$, $S_{\theta(i)}$: $\gamma_i \rightarrow \gamma_{i+1}$. We refer to γ_* as γ_{*1}. We have $\gamma_1 \sim_{(pts,\cup_i l_i)} \gamma_{*1}$ which implies $\gamma_1 \sim_{(pts \cup \cup_{j\neq\theta(1)} pts_j, l_{\theta(1)})} \gamma_{*1}$. Therefore by induction hypothesis, there exists γ_{*2} such that $S_{\theta(1)}$: $\gamma_{*1} \rightarrow \gamma_{*2}$ and $\gamma_2 \sim_{(pts_{\theta(1)}, l' \cup \cup_j l_{j\neq\theta(1)})} \gamma_{*2}$ which implies $\gamma_2 \sim_{(pts \cup \cup_{j\neq\theta(2)} pts_j, l_{\theta(2)})} \gamma_{*2}$. Therefore a simple induction on n proves the required. $\square$

5. Dead code elimination

This section introduces a type system for dead code elimination. Given a program and a set of variables whose values concern us at the end of the program, there may be some code in the program that has no effect on the values of these variables. Such code is called *dead code*. The type system presented here aims at optimizing structured parallel programs with pointer constructs via eliminating dead code. In the form of a type derivation, the type system associates each optimization with a proof for the soundness of the optimization. Optimizing a program may result in correcting it i.e. preventing it from aborting. Of course this happens if the removed dead code is the only cause of abortion.

The type system presented here has judgements of the form: S: $(pts\ ,l) \rightarrow (pts',l') \hookrightarrow S'$. The intuition is that S' optimizes S towards dead code elimination (and may be program correction). As mentioned early in many occasions, the derivation of such judgement provides a justification for the optimization process. The form of the judgement makes it apparent that the type system presented in this section is built on the type system for live-variables analysis.

Fig. 3 outlines an algorithm, *parallel-optimize*, that summarizes the optimization process. A pointer analysis that annotates the points of the input program with pointer information is the first step of the algorithm. This step takes the form of a post type derivation of S, in our type system for pointer analysis, using the bottom points-to type $\bot = \{x \mapsto \emptyset \| x \in Var\}$ as the pre type. Secondly, the algorithm refines the pointer information obtained in the first step via annotating the pointer types with type components for live-variables. Using our type systems for live-variables analysis, this is done via a pre type derivation of S for the set l', the set of variables whose values concerns us at the end of execution, as the post type. Finally, the information obtained so far is utilized in the third step to find S' via using the type system for dead code elimination proposed in this section. Applying this algorithm to the program on the left-hand side of Fig. 1 results in the program on the right-hand side of the same figure. The details of this application is a simple exercise.

The inference rules of our type system for dead code elimination are as follows:

$$\frac{x := e :\ pts\ \rightarrow\ pts' \quad x \notin l'}{x := e :\ (pts, l')\ \rightarrow\ (pts', l') \hookrightarrow skip}\ (:=^e_1)$$

$$\frac{x := e :\ pts\ \rightarrow\ pts' \quad x \in l'}{x := e :\ (pts, (l' \setminus \{x\}) \cup FV(e))\ \rightarrow\ (pts', l') \hookrightarrow x := e}\ (:=^e_2)$$

$$\frac{x \notin l'}{x := \&y :\ (pts, l')\ \rightarrow\ (pts[x \mapsto \{y'\}], l') \hookrightarrow skip}\ (:= \&^e_1)$$

$$\frac{}{skip :\ (pts, l)\ \rightarrow\ (pts, l) \hookrightarrow skip}$$

$$\frac{x \in l'}{x := \&y :\ (pts, l' \setminus \{x\}) \rightarrow (pts[x \mapsto \{y'\}], l') \hookrightarrow x := \&y}\ (:= \&^e_2)$$

$$\frac{x := *y : pts \rightarrow pts' \quad x \notin l'}{x := *y :\ (pts, l' \cup \{y\}) \rightarrow (pts', l') \hookrightarrow skip}\ (:= *^e_1)$$

$$\frac{x := *y : pts \rightarrow pts' \quad x \in l'}{x := *y :\ (pts, (l' \setminus \{x\}) \cup \{y, z | z' \in pts(y)\}) \rightarrow (pts', l') \hookrightarrow x := *y}\ (:= *^e_2)$$

$$\frac{*x := e : pts \rightarrow pts' \quad pts(x) \cap l = \emptyset}{*x := e : (pts, l' \cup \{x\}) \rightarrow (pts', l') \hookrightarrow skip} (*:=^e_1)$$

$$\frac{*x := e : pts \rightarrow pts' \quad pts(x) \cap l' \neq \emptyset}{*x := e : (pts, l' \cup \{x\} \cup FV(e)) \rightarrow (pts', l') \hookrightarrow *x := e} (*:=^e_1)$$

$$\frac{S_i : (pts \cup \cup_{j \neq i} pts_j, l_i) \rightarrow (pts_i, l' \cup \cup_{j \neq i} l_j) \hookrightarrow S'_i}{par\{\{S_1\},\ldots,\{S_n\}\} : (pts, \cup_i l_i) \rightarrow (\cup_i pts_i, l') \hookrightarrow par\{\{S'_1\},\ldots,\{S'_n\}\}} (par^e)$$

$$\frac{\begin{array}{l} par\{\{if\ b_1\ then\ S_1\ else\ skip\},\ldots,\{if\ b_n\ then\ S_n\ else\ skip\}\} : (pts,l) \rightarrow (pts',l') \\ \hookrightarrow par\{\{if\ b_1\ then\ S'_1\ else\ skip\},\ldots,\{if\ b_n\ then\ S'_n\ else\ skip\}\} \end{array}}{\begin{array}{l} par-if\{(b_1,S_1),\ldots,(b_n,S_n)\} : (pts,l) \rightarrow (pts',l') \\ \hookrightarrow par-if\{(b_1,S'_1),\ldots,(b_n,S'_n)\} \end{array}} (par-if^e)$$

$$\frac{S : (pts \cup pts', l) \rightarrow (pts', l' \cup l) \hookrightarrow S'}{par-for\{S\} : (pts,l) \rightarrow (pts',l') \hookrightarrow par-for\{S'\}} (par-for^e)$$

$$\frac{S_1 : (pts,l) \rightarrow (pts'',l'') \hookrightarrow S'_1 \quad S_2 : (pts'',l'') \rightarrow (pts',l') \hookrightarrow S'_2}{S_1;S_2 : (pts,l) \rightarrow (pts',l') \hookrightarrow S'_1;S'_2} (seq^e)$$

$$\frac{S_t : (pts,l) \rightarrow (pts',l') \hookrightarrow S'_t \quad S_f : (pts,l) \rightarrow (pts',l') \hookrightarrow S'_f}{if\ b\ then\ S_t\ else\ S_f : (pts, l \cup FV(b)) \rightarrow (pts',l') \hookrightarrow if\ b\ then\ S'_t\ else\} S'_f} (if^e)$$

$$\frac{l = l' \cup FV(b) \quad S_t : (pts,l') \rightarrow (pts,l) \hookrightarrow S'_t}{while\ b\ do\ S_t : (pts,l) \rightarrow (pts,l') \hookrightarrow while\ b\ do\ S'_t} (whl^e)$$

$$\frac{(pts'_1,l'_1) \leqslant (pts_1,l_1) \quad S : (pts_1,l_1) \rightarrow (pts_2,l_2) \hookrightarrow S' \quad (pts_2,l_2) \leqslant (pts'_2,l'_2)}{S : (pts'_1,l'_1) \rightarrow (pts'_2,l'_2) \hookrightarrow S'} (csq^e)$$

When optimizing programs it is important to guarantee that if (a) the original and optimized programs are executed in similar states, and (b) the original program ends at a state (rather than *abort*), then (a) the optimized program does not abort as well, and (b) the optimized program reaches a state similar to that reached by the original program. Indeed, this is guaranteed by the following theorem.

Theorem 4. *(Soundness) Suppose that* $S: (pts,l) \rightarrow (pts',l') \hookrightarrow S'$ *and* $\gamma \sim_{(pts,l)} \gamma_*$. *Then*

1. *If* $S: \gamma \rightsquigarrow \gamma'$, *then there exists a state* γ'_* *such that* $S' : \gamma_* \rightarrow \gamma'_*$ *and* $\gamma' \sim_{(pts',l')} \gamma'_*$.
2. *If* $S' : \gamma_* \rightarrow \gamma'_*$ *and* S *does not abort at* γ, *then there exists a state* γ' *such that* $S: \gamma \rightsquigarrow \gamma'$ *and* $\gamma' \sim_{(pts',l')} \gamma'_*$.

The proof of this theorem is by induction on the structure of type derivation and it follows smoothly from Theorem 3. More precisely Theorem 3 is used when $S' = S$. When $S' = skip$, we take $\gamma'_* = \gamma_*$ in 1. We note that the requirement of Theorem 3 that S does not abort at γ_* is guarantied when this theorem is called in the proof of Theorem 4.

6. Related work

6.1. Analysis of multithreaded programs

The analysis of multithreaded programs is an area that receives growing interest. It is a challenging area [27] as the presence of threading complicates the program analysis. The work in this area can be classified into two main categories. One category includes techniques that was designed specifically to optimize or correct multithreaded programs. The other category includes techniques whose scope was extended from sequential programs to multithreaded programs.

Under the first category mentioned above comes several directions of research. The purpose in the analysis of synchronization constructs [28,32] is to clarify how the synchronization actions apart executions of program segments. The result of this analysis can be used by compiler to conveniently add join-fork constructs. One problem of multithreading computing is deadlock which results from round waiting to gain resources. Researchers have developed various techniques for deadlock detection [9,30,31]. The situation when a memory location is accessed by two threads (one of them writes in the location) without synchronization is called data race. On direction of research in this category focuses on data race detection [15]. The analysis of multithreaded programs becomes even harder in the presence of a weak memory consistency model because such model does not guarantee that a write statement included in one thread is observed by other threads in the same order. However such model simplifies some issues on the hardware level. The work in this direction, like [5], aims at overcomes the drawbacks of using a simple consistency memory model.

Under the second category mentioned above comes several directions of research. One such direction is the using of flow-insensitive analysis techniques to analyze multithreaded programs [18,24]. Although flow-insensitive techniques are not very precise, some applications can afford that. Examples of program analyses whose techniques were extended to cover multithreaded programs are code motion [11], constant propagation [14], data flow for multithreaded programs with copy-in and copy-out memory semantics [10,17], and concurrent static single assignment form [13].

The problem with almost all the work refereed to above is that it does not apply to pointer programs. More precisely, for some of the work the application is possible only if we have the result of a pointer analysis for the input pointer program. The technique presented in this paper for optimizing multithreaded programs has the advantage of being simpler and more reliable than the optimization techniques refereed to above that would work in the presence of a pointer analysis.

6.2. Pointer analysis

The pointer analysis for sequential programs has been studied extensively for decades [7]. One way to classify the work in this area is according to properties of flow-sensitivity and context-sensitivity.

Flow-sensitive analyses [6,29,33], which are more natural than flow-insensitive to most applications, consider the order of program commands. Mostly these analyses perform an abstract interpretation of program using dataflow analysis to associate each program point with a points-to relation. Flow-insensitive analyses [1,2] do not consider the order of program commands. Typically the output of these analyses, which are performed using a constraint-based approach, is a points-to relation that is valid all over the program. Clearly the flow-sensitive approach is more precise but less efficient than the flow-insensitive one. Moreover flow-insensitive techniques can be used to analyze multithreaded programs.

The idea of context-sensitive approach [20,33] is to produce a points-to relation for the context of each call site of each procedure. On the other hand, the context-insensitive [16] pointer analysis produces one points-to relation for each procedure to cover contexts of all call sites. As expected the context-sensitive approach is more precise but less efficient than the context-insensitive one.

Although the problem of pointer analysis for sequential programs was studied extensively, a little effort was done towards a pointer analysis for multithreaded programs. In [25], a flow sensitive analysis for multithreaded programs was introduced. This analysis associates each program point with a triple of points-to relations. This in turn complicates the analysis and creates a sort of redundancy in the collected points-to information. Investigating the details of this approach and our work makes it apparent that our work is simpler and more accurate than this approach. Moreover our approach provides a proof for the correctness of the pointer analysis for each program. To the best of our knowledge, such proof is not known to be provided by any other existing approach.

6.3. Type systems in program analysis

The work in [3,12,22] is among the closest work to ours in the sense that it uses type systems to achieve the program analysis in a way similar to ours. The work in [26] can be seen as a special case of our work for the case of while language where there is no threading nor pointer constructs.

The work in [12] shows that a good deal of program analysis can be done using type systems. More precisely, it proves that for every analysis in a certain class of data-flow analyses, there exists a type system such that a program checks with a type if and only if the type is a supertype for the set resulting from running the analysis on the program. The type system in [19] and the flow-logic work in [22], which is used in [21] to study security of the coordinated systems, are very similar to [12]. For the simple while language, the work in [3] introduces type systems for constant folding and dead code elimination and also logically proves correctness of optimizations. The bidirectional data-flow analyses and their program optimizations are treated with type systems in [4]. Earlier, related work (with structurally-complex type systems) is [23].

To the best of our knowledge, our approach is the first attempt to use type systems to optimize multithreaded programs and associates every individual optimization with a justification for correctness.

References

[1] S. Adams, T. Ball, M. Das, S. Lerner, S.K. Rajamani, M. Seigle, W. Weimer, Speeding up dataflow analysis using flow-insensitive pointer analysis, in: M.V. Hermenegildo, G. Puebla (Eds.), SAS, Lecture Notes in Computer Science, vol. 2477, Springer, 2002, pp. 230–246.

[2] P. Anderson, D. Binkley, G. Rosay, T. Teitelbaum, Flow insensitive points-to sets, Information & Software Technology 44 (13) (2002) 743–754.

[3] N. Benton, Simple relational correctness proofs for static analyses and program transformations, in: N.D. Jones, X. Leroy (Eds.), POPL, ACM, 2004, pp. 14–25.

[4] M.J. Frade, A. Saabas, T. Uustalu, Bidirectional data-flow analyses, type-systematically, in: PEPM, ACM, 2009, pp. 141–150.

[5] I. Gelado, J. Cabezas, N. Navarro, J.E. Stone, S.J. Patel, mei W. Hwu, An asymmetric distributed shared memory model for heterogeneous parallel systems, in: J.C. Hoe, V.S. Adve (Eds.), ASPLOS, ACM, 2010, pp. 347–358.

[6] B. Hardekopf, C. Lin, Semi-sparse flow-sensitive pointer analysis, in: Z. Shao, B.C. Pierce (Eds.), POPL, ACM, 2009, pp. 226–238.

[7] M. Hind, Pointer analysis: haven't we solved this problem yet? in: PASTE, 2001, pp. 54–61.

[8] C.A.R. Hoare, An axiomatic basis for computer programming, Communications of the ACM 12 (10) (1969) 576–580.

[9] B.-C. Kim, S.-W. Jun, D.J. Hwang, Y.-K. Jun, Visualizing potential deadlocks in multithreaded programs, in: V. Malyshkin (Ed.), PaCT, Lecture Notes in Computer Science, vol. 5698, Springer, 2009, pp. 321–330.

[10] J.S. Kin, J.L. Pino, Multithreaded synchronous data flow simulation, in: DATE, IEEE Computer Society, 2003, pp. 11094–11095.

[11] J. Knoop, B. Steffen, Code motion for explicitly parallel programs, in: PPOPP, 1999, pp. 13–24.

[12] P. Laud, T. Uustalu, V. Vene, Type systems equivalent to data-flow analyses for imperative languages, Theor. Comput. Sci. 364 (3) (2006) 292–310.

[13] J. Lee, S.P. Midkiff, D.A. Padua, Concurrent static single assignment form and constant propagation for explicitly parallel programs, in: Z. Li, P.-C. Yew, S. Chatterjee, C.-H. Huang, P. Sadayappan, D.C. Sehr (Eds.), LCPC, Lecture Notes in Computer Science, vol. 1366, Springer, 1997, pp. 114–130.

[14] J. Lee, S.P. Midkiff, D.A. Padua, A constant propagation algorithm for explicitly parallel programs, International Journal of Parallel Programming 26 (5) (1998) 563–589.

[15] K. Leung, Z. Huang, Q. Huang, P. Werstein, Maotai 2.0: Data race prevention in view-oriented parallel programming, in: PDCAT, IEEE Computer Society, 2009, pp. 263–271.

[16] D. Liang, M. Pennings, M.J. Harrold, Extending and evaluating flow-insenstitive and context-insensitive points-to analyses for java, in: PASTE, 2001, pp. 73–79.

[17] M. Liu, W. Ji, X. Pu, J. Li, A parallel memory system model for multi-core processor, in: NAS, IEEE Computer Society, 2009, pp. 219–222.

[18] E. Lübbers, M. Platzner, Communication and synchronization in multithreaded reconfigurable computing systems, in: T.P. Plaks (Ed.), ERSA, CSREA Press, 2008, pp. 83–89.

[19] M. Naik, J. Palsberg, A type system equivalent to a model checker, ACM Transactions on Programming Languages and Systems 30 (5) (2008) 1–24.

[20] R. Nasre, K. Rajan, R. Govindarajan, U.P. Khedker, Scalable context-sensitive points-to analysis using multi-dimensional bloom filters, in: Z. Hu (Ed.), APLAS, Lecture Notes in Computer Science, vol. 5904, Springer, 2009, pp. 47–62.

[21] R.D. Nicola, D. Gorla, R.R. Hansen, F. Nielson, H.R. Nielson, C.W. Probst, R. Pugliese, From flow logic to static type systems for coordination languages, Sci. Comput. Program. 75 (6) (2010) 376–397.

[22] H.R. Nielson, F. Nielson, Flow logic: a multi-paradigmatic approach to static analysis, in: T.Æ. Mogensen, D.A. Schmidt, I.H. Sudborough (Eds.), The Essence of Computation, Lecture Notes in Computer Science, vol. 2566, Springer, 2002, pp. 223–244.

[23] J. Palsberg, P. O'Keefe, A type system equivalent to flow analysis, ACM Transactions on Programming Languages and Systems 17 (4) (1995) 576–599.

[24] Ruf, E., Effective synchronization removal for java, in: PLDI, 2000, pp. 208–218.

[25] R. Rugina, M.C. Rinard, Pointer analysis for structured parallel programs, ACM Transactions on Programming Languages and Systems 25 (1) (2003) 70–116.

[26] A. Saabas, T. Uustalu, Program and proof optimizations with type systems, Journal of Logic and Algebraic Programming 77

(1-2) (2008) 131–154 (The 16th Nordic Workshop on the Prgramming Theory (NWPT 2006)).

[27] V. Sarkar, Challenges in code optimization of parallel programs, in: O. de Moor, M.I. Schwartzbach (Eds.), CC, Lecture Notes in Computer Science, vol. 5501, Springer, 2009, p. 1.

[28] C. Tian, V. Nagarajan, R. Gupta, S. Tallam, Automated dynamic detection of busy–wait synchronizations, Softw., Pract. Exper. 39 (11) (2009) 947–972.

[29] J. Wang, X. Ma, W. Dong, H.-F. Xu, W. Liu, Demand-driven memory leak detection based on flow- and context-sensitive pointer analysis, J. Comput. Sci. Technol. 24 (2) (2009) 347–356.

[30] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, S.A. Mahlke, Gadara: Dynamic deadlock avoidance for multithreaded programs, in: R. Draves, R. van Renesse (Eds.), OSDI, USENIX Association, 2008, pp. 281–294.

[31] X. Xiao, J.J. Lee, A true o(1) parallel deadlock detection algorithm for single-unit resource systems and its hardware implementation, IEEE Transactions on Parallel and Distributed Systems 21 (1) (2010) 4–19.

[32] C. Xu, Y. Che, J. Fang, Z. Wang, Optimizing adaptive synchronization in parallel simulators for large-scale parallel systems and applications, in: CIT, IEEE Computer Society, 2010, pp. 131–138.

[33] H. Yu, J. Xue, W. Huo, X. Feng, Z. Zhang, Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code, in: A. Moshovos, J.G. Steffan, K.M. Hazelwood, D.R. Kaeli (Eds.), CGO, ACM, 2010, pp. 218–229.