Inferring Channel Buffer Bounds via Linear Programming

Tachio Terauchi[1] and Adam Megacz[2]

[1] Tohoku University
terauchi@ecei.tohoku.ac.jp
[2] University of California, Berkeley
megacz@cs.berkeley.edu

Abstract. We present a static analysis for inferring the maximum amount of buffer space used by a program consisting of concurrently running processes communicating via buffered channels. We reduce the problem to linear programming by casting the analysis as a fractional capability calculus system. Our analysis can reason about buffers used by multiple processes concurrently, and runs in time polynomial in the size of the program.

1 Introduction

We consider programs consisting of concurrently running processes communicating via buffered channels. Each process runs sequentially at its own speed, and synchronizes by communicating over channels. Communications are buffered in the sense that the messages may not be immediately sent to the receiver, but are held at some place. But holding messages costs buffer resources. If the buffers have a predetermined maximum size, unwanted behavior may happen if a process tries to send over a channel whose buffer is full. If the buffer is lossy, messages could get lost. Otherwise, it could block or change the sender process's control flow. This paper presents a static analysis for obtaining a conservative bound on channel buffers so that such behavior never happens, that is, channel buffers are used within their bounds. Such an analysis has application in determining a program's resource usage bound.

We cast our analysis as a *capability calculus*. The capability calculus is a static system originally proposed for reasoning about resources in sequential computation [2]. We use the extension of the capability calculus to channel communicating concurrent programs to allow capabilities to be passed at synchronization points [6]. We also use fractional capabilities [1, 5, 6] so that we can efficiently infer capabilities via linear programming.

Our analysis can automatically discover some non-trivial buffer bounds. For example, consider the program in Figure 1 consisting of two concurrently running processes communicating via the channels `foo` and `bar`, used to transmit integer values. The variables `i, j, m, n` are assumed to be initialized to some positive integers. Process 1 reads from the channel `bar` and stores the read value in

```
Process 1              Process 2
while i < m            while j < n
  bar?(x);               bar!(j);
  foo!(1);               foo?(y);
  foo!(i);               foo?(z);
  i := i + x             j := j + y + z
```

Fig. 1. Example.

variable `x`, writes twice to `foo`, and then updates the variable `i` and repeats if the loop condition is met. Process 2 writes once to `bar` and reads twice from `foo`, and then repeats if the loop condition is met. Buffer space to store only one integer is needed for the channel `bar`. This is because when process 2 is about to write to `bar` for the second time, process 1 must have already read the first integer from `bar` as process 2's write is preceded by the two reads from `foo` in the previous iteration, which in turn were written by process 1 after the read from `bar`. The same argument holds by induction for the subsequent iteration of the loop. Similarly, the program only needs buffer space to store two integers for the channel `foo`. Our analysis is able to automatically infer these optimal bounds.

The rest of the paper is organized as follows. Section 2 introduces the syntax of the simple concurrent language we use to describe the analysis. Section 3 defines the operational semantics of the language and formally defines what it means for a program to run within a buffer bound. Section 4 presents the capability calculus which statically guarantees that a program runs within a buffer bound. Section 5 presents the analysis algorithm as a type inference algorithm for the capability calculus. Section 6 discusses limitations of our work. Section 7 discusses related work. Section 8 concludes.

2 The Simple Concurrent Language

We focus on the simple concurrent language shown in Figure 2. The language is essentially the simple imperative language WHILE extended with concurrency primitives. Formally, a program, p, is a parallel composition of finitely many processes. A process, $i.s$, is a sequential statement s prefixed by a process index i. A sequential statement consists of the usual imperative features as well as primitives for buffered communications. Here, $e_1!(e_2)$ means writing the value of e_2 to the buffered channel e_1, and $e?(x)$ means storing the value read from the channel e in variable x. The variables are process-local, and so the only means of communication are channel reads and writes. We use meta-variables x, x', etc. for variables and c, c', etc. for channels. Channels are first class and can be used as values, that is, they can be assigned to variables or written to channels. Binary integer operations such as $+, -, \times, \leq$, etc., are ranged over by the symbol *op*.

$$
\begin{array}{lll}
p ::= & i.s & (\textit{process}) \\
\mid & p_1 \mid\mid p_2 & (\textit{parallel composition}) \\
s ::= & s_1; s_2 & (\textit{sequential composition}) \\
\mid & \texttt{skip} & (\textit{skip}) \\
\mid & \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 & (\textit{branch}) \\
\mid & \texttt{while } e \texttt{ do } s & (\textit{loop}) \\
\mid & x \texttt{ := } e & (\textit{assignment}) \\
\mid & e_1\texttt{!}(e_2) & (\textit{channel write}) \\
\mid & e\texttt{?}(x) & (\textit{channel read}) \\
e ::= & c & (\textit{channel constant}) \\
\mid & x & (\textit{local variable}) \\
\mid & n & (\textit{integer constant}) \\
\mid & e_1 \; op \; e_2 & (\textit{integer operation})
\end{array}
$$

Fig. 2. The syntax of the simple concurrent language.

To keep the presentation to the novel features of the analysis, this simple language lacks the ability to create processes and channels dynamically, but it is easy to extend the analysis to handle dynamic creation of processes and channels by borrowing the techniques from [3, 6].

3 Operational Semantics

We define the following mathematical convention. Given a mapping (i.e., a set-theoretic function) f, $f[a \mapsto b]$ is a mapping such that $f[a \mapsto b](a) = b$ and $f[a \mapsto b](a') = f(a')$ for $a' \neq a$.

The operational semantics of the language is defined as a series of reductions from states to states. A state is represented by the triple (B, S, p) where B is a *buffer* and S is a *store*.

A store is a mapping from process index to process store. A *process store* a mapping from variables to values. We use symbols h, h', etc. to denote a process store. Values are subset of expressions (e) defined as follows.

$$v ::= c \mid n$$

Figure 3 shows the evaluation rules. Expressions are evaluated entirely locally. Their evaluation relation are of the form $(h, e) \Downarrow v$ and defined by the rules **Chan**, **Int**, **Var**, and **Op**. Here, $[\![op]\!]$ is the standard semantics of the binary operator *op*. The sequential composition operator ; is associative. Also, we let `skip` be a ; identity, that is, $s = s; \texttt{skip} = \texttt{skip}; s$. The parallel composition operator || is commutative and associative, e.g., $p_1 \mid\mid p_2 \mid\mid p_3 = p_2 \mid\mid p_3 \mid\mid p_1$. Note that the process reduction rules only reduce the left-most process, and so we rely on process re-ordering to reduce other processes. We assume that the process indices are disjoint in any program p. **If1**, **If2**, **While1**, and **While2** do not involve channel communication and are self-explanatory. **Assign** is also a process-local reduction because variables are local.

$$\frac{}{(h, c) \Downarrow c}\ \textbf{Chan} \qquad \frac{}{(h, n) \Downarrow n}\ \textbf{Int}$$

$$\frac{}{(h, x) \Downarrow h(x)}\ \textbf{Var} \qquad \frac{(h, e_1) \Downarrow n_1 \quad (h, e_2) \Downarrow n_2}{(h, e_1\ op\ e_2) \Downarrow n_1\ [\![op]\!]\ n_2}\ \textbf{Op}$$

$$\frac{(S(i), e) \Downarrow n \qquad n \neq 0}{(B, S, i.(\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2); s \mid\mid p) \rightarrow (B, S, i.s_1; s \mid\mid p)}\ \textbf{If1}$$

$$\frac{(S(i), e) \Downarrow 0}{(B, S, i.(\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2); s \mid\mid p) \rightarrow (B, S, i.s_2; s \mid\mid p)}\ \textbf{If2}$$

$$\frac{(S(i), e) \Downarrow n \qquad n \neq 0}{(B, S, i.(\texttt{while}\ e\ \texttt{do}\ s_1); s \mid\mid p) \rightarrow (B, S, i.s_1; (\texttt{while}\ e\ \texttt{do}\ s_1); s \mid\mid p)}\ \textbf{While1}$$

$$\frac{(S(i), e) \Downarrow 0}{(B, S, i.(\texttt{while}\ e\ \texttt{do}\ s_1); s \mid\mid p) \rightarrow (B, S, i.s \mid\mid p)}\ \textbf{While2}$$

$$\frac{(S(i), e) \Downarrow v \qquad S' = S[i \mapsto S(i)[x \mapsto v]]}{(B, S, i.x\ \texttt{:=}\ e; s \mid\mid p) \rightarrow (B, S', i.s \mid\mid p)}\ \textbf{Assign}$$

$$\frac{(S(i), e_1) \Downarrow c \qquad (S(i), e_2) \Downarrow v \qquad B' = B.write(c, v)}{(B, S, i.e_1\,\texttt{!}(e_2); s \mid\mid p) \rightarrow (B', S, i.s \mid\mid p)}\ \textbf{Write}$$

$$\frac{(S(i), e) \Downarrow c \qquad (B', v) = B.read(c) \qquad S' = S[i \mapsto S(i)[x \mapsto v]]}{(B, S, i.e\texttt{?}(x); s \mid\mid p) \rightarrow (B', S', i.s \mid\mid p)}\ \textbf{Read}$$

Fig. 3. The operational semantics of the simple concurrent language.

Write and **Read** handle communications over channels. We write $B.write(c, v)$ for the buffer B after v is written to the channel c, and $B.read(c)$ for the pair (B', v) where v is the value read from channel c and B' is the buffer after the read.

Formally, a buffer B is a mapping from channels to buffer contents. We model buffer contents as a bag of values. Buffer writes and reads are defined as follows.

$$\begin{array}{l} B.write(c, v) = B[c \mapsto B(c) \uplus \{v\}] \\ B.read(c) = (B[c \mapsto S], v) \qquad \text{if } B(c) = S \uplus \{v\} \end{array}$$

Here, $\uplus$ denotes bag union, e.g., $\{v\} \uplus \{v\} = \{v, v\}$. Note that we are not concerned about the order of values written/read to/from a buffer, and so to allow maximum generality, we model a buffer as a bag of values from which an arbitrary value can be read at a channel read provided that the bag is non-empty.

The operational semantics allows arbitrary many values to be stored in a buffer. In practice, buffers may be bounded due to physical resource constraints. Exactly what happens if a sender tries to write to a full buffer is outside of the scope of the paper. The goal of the analysis is to infer buffer bounds to ensure

that such behavior never occurs. In contrast, a receiver is allowed to wait on an empty buffer, allowing the processes to synchronize over a channel.

For simplicity, we assume that every value has the same size and occupies the same amount of space in the buffers. We write $P \to^* Q$ for zero or more reduction steps from the state P to the state Q. We now formally define what it means for a program to run within a buffer bound.

Definition 1. *We say that the buffer bound of c in P is within n if for any (B, S, p) such that $P \to^* (B, S, p)$, $|B(c)| \leq n$.*

4 The Capability Calculus

Our analysis returns a buffer bound for each channel in the program. To this end, we design a capability calculus such that given a state P, we can obtain a buffer bound for each channel in P from the derivation for P in the calculus.

The capability calculus is a kind of a type system. The types are defined as follows.

$$\begin{array}{rll} \tau ::= & ch(\rho, \tau, \Psi) & (\textit{channels}) \\ \mid & \textit{int} & (\textit{integers}) \end{array}$$

The type $ch(\rho, \tau, \Psi)$ denotes a type of a channel used to send and receive values of the type τ. Here, ρ is the *handle* of the channel. Let **Handles** be the set of channel handles. Symbols Ψ, Ψ', etc. represent *capability mappings*. A capability mapping is a function from **Handles** to non-negative rational numbers augmented with ∞, that is, $\mathbb{Q}^+ \cup \{0, \infty\}$. We use the ordering $q \leq \infty$ for all $q \in \mathbb{Q}^+ \cup \{0, \infty\}$, and the following arithmetic relation: $q + \infty = \infty$, $q \times \infty = \infty$ for $q \neq 0$, and $0 \times \infty = 0$.

We say that Ψ such that $\Psi(\rho) = q$ has q amount of ρ. We often refer to Ψ itself as "capabilities", with the understanding that we mean the amount of capabilities in Ψ. Capabilities are conceptual, that is, capabilities only exist in the static type system world and do not appear in the dynamic semantics. Conceptually, each process holds some amount of capabilities representing the amount of buffer space available for its use. For instance, a process holding capabilities Ψ may write $\Psi(\rho)$ many values to the buffers for channels with the handle ρ. The capability mapping appearing in a channel type represent the capabilities that are passed when communicating over that channel. That is, when two processes communicate over a channel having the type $ch(\rho, \tau, \Psi)$, the sender process passes the capabilities Ψ to the receiver process.

We define arithmetic operations over capabilities. The addition and subtraction of capability mappings are defined point-wise as $\Psi + \Psi' = \lambda\rho.\Psi(\rho) + \Psi'(\rho)$ and $\Psi - \Psi' = \lambda\rho.\Psi(\rho) - \Psi'(\rho)$. Because capabilities must be non-negative, $\Psi - \Psi'$ is undefined if $\Psi(\rho) < \Psi'(\rho)$ for some ρ. We define the relation $\Psi \leq \Psi'$ point-wise as $\forall \rho \in \textbf{Handles}.\Psi(\rho) \leq \Psi'(\rho)$. For convenience, we let $\mathit{0}$ denote a constant capability mapping that maps all handles to 0, that is, $\mathit{0} = \lambda\rho.0$. Therefore, for example, $\mathit{0}[\rho \mapsto 1]$ is a capability mapping that maps ρ to 1 and ρ' to 0 for all $\rho' \neq \rho$.

$$\frac{}{\Gamma \vdash c : \Gamma(c)}\ \textbf{CHAN} \qquad \frac{}{\Gamma \vdash n : int}\ \textbf{INT}$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ \textbf{VAR} \qquad \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1\ op\ e_2 : int}\ \textbf{OP}$$

$$\frac{}{\Gamma, \Psi \vdash \texttt{skip} : \Psi}\ \textbf{SKIP} \qquad \frac{\Gamma \vdash e : \Gamma(x)}{\Gamma, \Psi \vdash x := e : \Psi}\ \textbf{ASSIGN}$$

$$\frac{\Gamma, \Psi \vdash s_1 : \Psi_1 \quad \Gamma, \Psi_1 \vdash s_2 : \Psi_2}{\Gamma, \Psi \vdash s_1; s_2 : \Psi_2}\ \textbf{SEQ}$$

$$\frac{\Gamma \vdash e : int \quad \Psi' \leq \Psi_1 \quad \Psi' \leq \Psi_2 \quad \Gamma, \Psi \vdash s_1 : \Psi_1 \quad \Gamma, \Psi \vdash s_2 : \Psi_2}{\Gamma, \Psi \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 : \Psi'}\ \textbf{IF}$$

$$\frac{\Gamma \vdash e : int \quad \Gamma, \Psi' \vdash s : \Psi'' \quad \Psi' \leq \Psi \quad \Psi' \leq \Psi''}{\Gamma, \Psi \vdash \texttt{while}\ e\ \texttt{do}\ s : \Psi'}\ \textbf{WHILE}$$

$$\frac{\Gamma \vdash e : ch(\rho, \Gamma(x), \Psi')}{\Gamma, \Psi \vdash e?(x) : \Psi + \Psi' + 0[\rho \mapsto 1]}\ \textbf{READ}$$

$$\frac{\Gamma \vdash e : ch(\rho, \tau, \Psi') \quad \Gamma \vdash e' : \tau}{\Gamma, \Psi \vdash e!(e') : \Psi - \Psi' - 0[\rho \mapsto 1]}\ \textbf{WRITE}$$

Fig. 4. The type checking rules.

Figure 4 shows the type checking rules. The judgements for expressions are of the form $\Gamma \vdash e : \tau$, where Γ is a type environment mapping variables and channels to their types. The rules **VAR**, **CHAN**, **INT**, and **OP** type expressions and are self-explanatory.

The type judgements for the statements are of the form $\Gamma, \Psi \vdash s : \Psi'$, where Ψ is the capabilities before the execution of s, and Ψ' is the capabilities after the execution of s. **SKIP**, **SEQ**, and **ASSIGN** are self-explanatory. **IF** ensures that the capabilities at the branch join point cannot exceed the capabilities after the then branch or the else branch. **WHILE** is similar to **IF**.

In **READ**, the hypothesis ensures that type of the received value agrees with the type of the variable where the value is going to be stored. In the conclusion of **READ**, the capabilities Ψ' passed from the sender is added to the capabilities held by the process. In addition, because a read frees a buffer space, we gain a single buffer space, and so we add the capability $0[\rho \mapsto 1]$.

WRITE passes Ψ' to the receiver, and thus the capabilities Ψ' is subtracted in the conclusion of the rule. The subtraction of capabilities is defined as $\Psi_1 - \Psi_2 = \Psi_3$ iff $\Psi_3 + \Psi_2 = \Psi_1$. In addition, because a write uses a buffer space, we express this by subtracting $0[\rho \mapsto 1]$ in the conclusion. Note that the non-negativity assumption of capabilities implies that $\Psi(\rho) \geq 1$.

We define some notational shortcuts. Let $writeSend(ch(\rho, \tau, \Psi)) = \Psi$ and $hdl(ch(\rho, \tau, \Psi)) = \rho$. Let $HC_B(\rho, \Gamma)$ be the subset of the domain of B having the

handle ρ, that is,

$$HC_B(\rho, \Gamma) = \{c \in dom(B) \mid hdl(\Gamma(c)) = \rho\}$$

Note that $|HC_B(\rho, \Gamma)| > 1$ means that multiple channels have the same handle ρ.

We write $\Gamma \vdash B(c)$ to mean that the buffer $B(c)$ is well-typed, that is, for each $v \in B(c)$, $\Gamma \vdash v : \tau$, where $\Gamma(c) = ch(\rho, \tau, \Psi)$ for some ρ, Ψ. We write $\Gamma \vdash h$ to mean that the process store h is well-typed, that is, $\Gamma \vdash h(x) : \Gamma(x)$ for each $x \in dom(h)$. Because variables are process local, without loss of generality, we assume that each process uses a disjoint set of variables.

Definition 2 (Well-typed State). *We write*

$$\Gamma, \Psi_1, \ldots, \Psi_n, \Psi_B \vdash (B, S, i_1.s_1 || \ldots || i_n.s_n)$$

if

(1) For each channel $c \in dom(B)$, $\Gamma \vdash B(c)$.
(2) For each i_j, $\Gamma \vdash S(i_j)$.
(3) For each s_j, $\Gamma, \Psi_j \vdash s_j : \Psi'_j$ for some Ψ'_j.
(4) $\Psi_B = \sum_{c \in dom(B)} |B(c)| \times writeSend(\Gamma(c))$.

In (4), $m \times \Psi$ is defined as $\lambda\rho.m \times \Psi(\rho)$.

For simplicity, we have used simple types so that some programs are untypable (for instance, a program that uses integers as channels). But it is easy to extend the system with sum types and recursive types so that all programs become typable [4].

We now state the main result of this section which says that a well-typed program runs within buffer bounds that can be obtained from its type derivation.

Theorem 1. *Suppose $\Gamma, \Psi_1, \ldots, \Psi_n, \Psi_B \vdash (B, S, p)$. Suppose $hdl(\Gamma(c)) = \rho$. Let $\Psi_p = \Psi_B + \sum_{j=1}^{n} \Psi_j$. Then the buffer bound of c in (B, S, p) is within $\Psi_p(\rho) + \sum_{c' \in HC_B(\rho,\Gamma)} |B(c')|$.*

The key steps of the proof appear in the appendix.

4.1 Example

Recall the following program from Section 1. Let us call this program p.

```
1.while i < m do (bar?(x); foo!(1); foo!(i); i := i + x) ||
2.while j < n do (bar!(j); foo?(y); foo?(z); j := j + y + z)
```

Let B be an empty buffer, that is, $B(\texttt{foo}) = B(\texttt{bar}) = \emptyset$. Let S be a store such that $S(1)$ maps $\texttt{i}, \texttt{m}, \texttt{x}$ to some integer and $S(2)$ maps $\texttt{j}, \texttt{n}, \texttt{y}, \texttt{z}$ to some integer.

Let

$$\begin{aligned}\Gamma = \{&\texttt{i} \mapsto \mathit{int}, \texttt{j} \mapsto \mathit{int}, \texttt{m} \mapsto \mathit{int}, \texttt{n} \mapsto \mathit{int},\\ &\texttt{x} \mapsto \mathit{int}, \texttt{y} \mapsto \mathit{int}, \texttt{z} \mapsto \mathit{int},\\ &\texttt{foo} \mapsto \mathit{ch}(\rho_{foo}, \mathit{int}, 0[\rho_{bar} \mapsto 0.5]),\\ &\texttt{bar} \mapsto \mathit{ch}(\rho_{bar}, \mathit{int}, 0[\rho_{foo} \mapsto 2])\}\end{aligned}$$

$$\begin{aligned}\Psi_1 &= \Psi_B = 0\\ \Psi_2 &= 0[\rho_{foo} \mapsto 2][\rho_{bar} \mapsto 1]\end{aligned}$$

Then, we have $\Gamma, \Psi_1, \Psi_2, \Psi_B \vdash (B, S, p)$. The type of `foo` indicates that whenever process 2 reads from `foo`, 0.5 amount of capability for `bar` is passed to process 2. Therefore, by reading `foo` twice, process 2 gains $0.5 + 0.5 = 1$ buffer space for `bar`. Likewise, `bar`'s type says that reading `bar` once begets two buffer space for `foo`.

Let $\Psi_p = \Psi_1 + \Psi_2 + \Psi_B$. Note that $\Psi_p(\rho_{foo}) = 2$ and $\Psi_p(\rho_{bar}) = 1$, indicating that the buffer bound of `foo` is 2 in (B, S, p) and the buffer bound of `bar` is 1 in (B, S, p). As argued in Section 1, these are the optimal bounds for the program.

5 Analysis Algorithm

Intuitively, the analysis algorithm is a type inference algorithm for the type system presented in Section 4. Because there are multiple type derivations possible for a program, we would like to obtain a derivation that gives the smallest buffer bound for each channel. Our strategy is to reduce the problem to linear programming such that the buffer bound appears as the objective function to be minimized.

The analysis is separated in two phases. Informally, the first phase infers everything about the type derivation except for the amount of capabilities. The second phase uses linear programming to find the minimum amount of capabilities required to complete the type derivation.

5.1 Phase 1

The first phase is mostly a standard type-based analysis based on unification constraints, generating capability constraints on the side. Figure 5 shows the constraint generation rules. Here, α's are type variables, ϱ's are channel handle variables, and φ's are capability mapping variables. The inference rules are straightforward constraint-based implementation of the type checking rules in Figure 4.

The inference judgement for expressions, $\Delta \vdash e : \alpha; C$, is read "given the environment Δ, e is inferred to have the type α with the set of constraints C." The inference judgement for statements, $\Delta, \varphi \vdash s : \varphi'; C$ is read "given environment Δ, s is inferred to have the *pre-capability* φ and the *post-capability* φ' with the set of constraints C."

$$\frac{\alpha, \varrho, \varphi \text{ fresh}}{\Delta \vdash c : \Delta(c); \{ch(\varrho, \alpha, \varphi) = \Delta(c)\}}\ \textbf{CHAN}$$

$$\frac{\alpha \text{ fresh}}{\Delta \vdash n : \alpha; \{\alpha = int\}}\ \textbf{INT} \qquad \frac{}{\Delta \vdash x : \Delta(x); \emptyset}\ \textbf{VAR}$$

$$\frac{\Delta \vdash e_1 : \alpha_1; C_1 \qquad \Delta \vdash e_2 : \alpha_2; C_2 \qquad \alpha_3 \text{ fresh}}{\Delta \vdash e_1\ op\ e_2 : \alpha_3; C_1 \cup C_2 \cup \{\alpha_1 = \alpha_2 = \alpha_3 = int\}}\ \textbf{OP}$$

$$\frac{\varphi \text{ fresh}}{\Delta, \varphi \vdash \texttt{skip} : \varphi; \emptyset}\ \textbf{SKIP}$$

$$\frac{\Delta \vdash e : \alpha; C \qquad \varphi \text{ fresh}}{\Delta, \varphi \vdash x\ \texttt{:=}\ e : \varphi; C \cup \{\alpha = \Delta(x)\}}\ \textbf{ASSIGN}$$

$$\frac{\Delta, \varphi_1 \vdash s_1 : \varphi_1'; C_1 \qquad \Delta, \varphi_2 \vdash s_2 : \varphi_2'; C_2}{\Delta, \varphi_1 \vdash s_1; s_2 : \varphi_2'; C_1 \cup C_2 \cup \{\varphi_1' = \varphi_2\}}\ \textbf{SEQ}$$

$$\frac{\Delta \vdash e : \alpha; C \qquad \Delta, \varphi_1 \vdash s_1 : \varphi_1'; C_1 \qquad \Delta, \varphi_2 \vdash s_2 : \varphi_2'; C_2 \qquad \varphi, \varphi' \text{ fresh}}{\begin{array}{l}\Delta, \varphi \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 : \varphi'; \\ \qquad C \cup C_1 \cup C_2 \cup \{\alpha = int, \varphi_1 = \varphi_2 = \varphi, \varphi' \leq \varphi_1', \varphi' \leq \varphi_2'\}\end{array}}\ \textbf{IF}$$

$$\frac{\Delta \vdash e : \alpha; C \qquad \Delta, \varphi' \vdash s : \varphi''; C' \qquad \varphi \text{ fresh}}{\Delta, \varphi \vdash \texttt{while}\ e\ \texttt{do}\ s : \varphi'; C \cup C' \cup \{\alpha = int, \varphi' \leq \varphi, \varphi' \leq \varphi''\}}\ \textbf{WHILE}$$

$$\frac{\Delta \vdash e : \alpha; C \qquad \varrho, \varphi, \varphi', \varphi'' \text{ fresh}}{\Delta, \varphi \vdash e\texttt{?}(x) : \varphi''; C \cup \{\alpha = ch(\varrho, \Delta(x), \varphi'), \varphi'' = \varphi + \varphi' + 0[\varrho \mapsto 1]\}}\ \textbf{READ}$$

$$\frac{\Delta \vdash e : \alpha; C \qquad \Delta \vdash e' : \alpha'; C' \qquad \varrho, \varphi, \varphi', \varphi'' \text{ fresh}}{\begin{array}{l}\Delta, \varphi \vdash e\texttt{!}(e') : \varphi''; \\ \qquad C \cup C' \cup \{\alpha = ch(\varrho, \alpha', \varphi'), \varphi'' = \varphi - \varphi' - 0[\varrho \mapsto 1]\}\end{array}}\ \textbf{WRITE}$$

Fig. 5. The type inference rules.

We initialize Δ such that each $\Delta(x)$ and each $\Delta(c)$ is a fresh type variable. We visit each AST node (expressions and statements) in a bottom up manner to build the set of constraints.

The resulting set of constraints contains two kinds of constraints:

(a) Type unification constraints: $\sigma = \sigma'$
(b) Capability inequality constraints: $\phi \leq \phi'$

where

$$\sigma ::= \alpha \mid ch(\varrho, \alpha, \varphi) \mid int$$
$$\phi ::= \varphi \mid 0[\varrho \mapsto 1] \mid \phi + \phi \mid \phi - \phi$$

Note that an equality constraint $\phi = \phi'$ can expressed by inequality constraints $\phi \leq \phi'$ and $\phi' \leq \phi$. The constraints of the kind (a) can be resolved by the standard unification algorithm, which may create more constraints of the kind

(b). In addition, it creates constraints of the form $\varrho = \varrho'$, which can also be resolved by the standard unification algorithm. This leaves us with a set of constraints of the kind (b).

5.2 Phase 2

The second phase of the algorithm finds a satisfying solution to the remaining constraints generated in the first phase. In general, there can be more than one solution to these constraints. We find the minimum solution as follows. Let $p = i_1.s_1 \parallel \ldots \parallel i_n.s_n$ be the program being analyzed. Phase 1 returns pre-capability φ_j for each process s_j such that $\Delta, \varphi_j \vdash s_j : \varphi'_j; C_j$. We create a fresh capability mapping variable φ_p and add the constraint $\varphi_p = \sum_{j=1}^{n} \varphi_j$.

Next, for each ϱ (that is, its equivalence class obtained via the unification in phase 1), we instantiate a linear programming problem using the remaining constraints together with the constraint $\varphi_p = \sum_{i=1}^{n} \varphi_i$. More precisely, each constraint mapping variable φ is instantiated as a linear programming variable $\varphi(\varrho)$, and $\mathit{0}[\varrho' \mapsto 1]$ is replaced by 1 if $\varrho' = \varrho$ and by 0 otherwise. We also add constraints $\varphi(\varrho) \geq 0$ to ensure that each capability mapping is non-negative. The objective function to minimize is $\varphi_p(\varrho)$. For any solution to the set of constraints, $\varphi_p(\varrho)$ is a valid buffer bound on the channel with the handle ϱ, and so minimizing $\varphi_p(\varrho)$ gives us the best possible buffer bound for the analysis.

We state the correctness of the analysis algorithm. We use the symbol η to denote a *constraint solution*, which is a sorted substitution mapping type variables to types, channel handle variables to channel handles, and capability mapping variables to capability mappings. A constraint solution becomes a mapping from σ, Δ, and ϕ in the obvious way (we let $\eta(\mathit{0}[\varrho \mapsto 1]) = \mathit{0}[\eta(\varrho) \mapsto 1]$).

Definition 3. *We write $\eta \models C$ ("η solves C") if*

- *for each $\sigma = \sigma' \in C$, $\eta(\sigma) = \eta(\sigma')$.*
- *for each $\phi \leq \phi' \in C$, $\eta(\phi) \leq \eta(\phi')$.*

Lemma 1.

- *If $\Delta \vdash e : \alpha; C$ and $\eta \models C$, then $\eta(\Delta) \vdash e : \eta(\alpha)$.*
- *If $\Delta, \varphi \vdash s : \varphi'; C$ and $\eta \models C$, then $\eta(\Delta), \eta(\varphi) \vdash s : \eta(\varphi)$.*

Proof. By induction on the type derivation.

Theorem 2 (Soundness). *Let $p = i_1.s_1 \parallel \ldots \parallel i_n.s_n$. Suppose*

$$\eta \models \{\varphi_p = \sum_{j=1}^{n} \varphi_j\} \cup \bigcup_{j=1}^{n} C_j$$

where $\Delta, \varphi_j \vdash s_j : \varphi'_j; C_j$ for each s_j. Let $P = (B, S, p)$ such that B is an empty buffer (i.e., $B(c) = \emptyset$ for all channels c) and S is a store such that $\eta(\Delta) \vdash S(i_j)$ for each i_j, then the buffer bound of c in P is within $\eta(\varphi_p)(\rho)$, where $\rho = \mathit{hdl}(\eta(\Delta)(c))$.

Proof. Straightforward from Lemma 1 and Theorem 1.

We have implemented a prototype of the analysis algorithm, available at http://research.cs.berkeley.edu/project/cccd-impl.

5.3 Analysis of The Algorithm

Linear programming is one of the most well studied problems in computer science. Algorithms with both good theoretical complexity and practical running times are known. The instance of linear programming problem in phase 2 can be solved in time polynomial in the size of the constraints by algorithms such as interior points methods.

Therefore, the complexity of the algorithm is bound by the time phase 1 takes to generate the capability constraints, which is polynomial for our simple concurrent language. In general, the complexity will increase if we include more complex programming constructs such as data structures and functions if we stick with the simple types. But this can be avoided by incorporating sum types and recursive types [4].

5.4 Example

We demonstrate the algorithm on the running example.

$$\begin{array}{l} 1.\texttt{while i} < \texttt{m do (bar?(x); foo!(1); foo!(i); i := i} + \texttt{x)} \;||\\ 2.\texttt{while j} < \texttt{n do (bar!(j); foo?(y); foo?(z); j := j} + \texttt{y} + \texttt{z)} \end{array}$$

Suppose that the following environment Δ was inferred in the first phase.

$$\begin{array}{l} \Delta(\texttt{i}) = \Delta(\texttt{j}) = \Delta(\texttt{m}) = \Delta(\texttt{n}) = \mathit{int}\\ \Delta(\texttt{x}) = \Delta(\texttt{y}) = \Delta(\texttt{z}) = \mathit{int}\\ \Delta(\texttt{foo}) = \mathit{ch}(\varrho_{foo}, \mathit{int}, \varphi_{foo})\\ \Delta(\texttt{bar}) = \mathit{ch}(\varrho_{bar}, \mathit{int}, \varphi_{bar}) \end{array}$$

The capability constraints generated from analyzing process 1 are as follows (after some simplification).

$$\begin{array}{l} \varphi_{entr1} \leq \varphi_{exit1}\\ \varphi_{temp11} = \varphi_{entr1} + \varphi_{bar} + 0[\varrho_{bar} \mapsto 1]\\ \varphi_{temp12} = \varphi_{temp11} - \varphi_{foo} - 0[\varrho_{foo} \mapsto 1]\\ \varphi_{exit1} = \varphi_{temp12} - \varphi_{foo} - 0[\varrho_{foo} \mapsto 1] \end{array}$$

Here, φ_{entr1} is the capabilities at the while loop entry, φ_{exit1} is the capabilities at the loop exit, φ_{temp11} is the capabilities after the read `bar?(x)`, and φ_{temp12} is the capabilities after the write `foo!(1)`. The capability constraints generated from analyzing process 2 are as follows (after some simplification).

$$\begin{array}{l} \varphi_{entr2} \leq \varphi_{exit2}\\ \varphi_{temp21} = \varphi_{entr2} - \varphi_{bar} - 0[\varrho_{bar} \mapsto 1]\\ \varphi_{temp22} = \varphi_{temp21} + \varphi_{foo} + 0[\varrho_{foo} \mapsto 1]\\ \varphi_{exit2} = \varphi_{temp22} + \varphi_{foo} + 0[\varrho_{foo} \mapsto 1] \end{array}$$

Here, φ_{entr2} is the capabilities at the while loop entry, φ_{exit2} is the capabilities at the loop exit, φ_{temp21} is the capabilities after `bar!(j)`, and φ_{temp22} is the capabilities after `foo?(y)`.

The capabilities to minimize is $\varphi_p = \varphi_{entr1} + \varphi_{entr2}$, or more precisely, $\varphi_p(\varrho_{foo})$ and $\varphi_p(\varrho_{bar})$. For $\varphi_p(\varrho_{bar})$, this reduces to solving the following linear programming instance.

$$\begin{array}{c} \text{minimize } entr_1 + entr_2 \\ \begin{array}{ll} exit_1 \geq entr_1 & exit_2 \geq entr_2 \\ temp_{11} = entr_1 + bar + 1 & temp_{21} = entr_2 - bar - 1 \\ temp_{12} = temp_{11} - foo & temp_{22} = temp_{21} + foo \\ exit_1 = temp_{12} - foo & exit_2 = temp_{22} + foo \end{array} \end{array}$$

We also add the constraint $a \geq 0$ for each linear programming variable a appearing above. The minimum solution is attained at

$$\begin{array}{l} \{entr_1 = 0, entr_2 = 1, bar = 0, foo = 0.5, \\ \quad temp_{11} = 1, temp_{12} = 0.5, exit_1 = 0, \\ \quad exit_2 = 1, temp_{21} = 0, temp_{22} = 0.5\} \end{array}$$

This gives us the bound $entr_1 + entr_2 = 1$. Similarly, solving for the minimum $\varphi_p(\varrho_{foo})$ gives us the bound 2 for `foo`.

6 Limitations

Our analysis cannot infer a finite buffer bound for channels written in a (reachable) loop whose capabilities cannot be "balanced" at the loop exit. Consider the following program.

$$1.\texttt{i := 0}; \texttt{while i} < 3 \texttt{ do } (\texttt{c!}(0); \texttt{i := i} + 1)$$

Clearly, the buffer bound for the channel `c` is 3. But note that the **WHILE** rule in Figure 4 requires the capabilities at the end of the loop to be greater than that of the start, and this is not possible for this loop due to `c!(0)`. This manifests in the analysis as ∞ returned as the bound (that is, there exists no finite solution to the linear programming instance). This implies that any loop that makes an "unbalanced send" must be unrolled prior to the analysis. This is actually an instance of the analysis's insensitivity to branch conditions. The issue just becomes most pronounced for loops.

Also, because of its simple flow&path-insensitive unification-based nature, our analysis may equate different channels when channels are used as values (e.g., stored in variables and passed as messages). This leads to different channels sharing the same buffer in the analysis. For example, analyzing the program below, the analysis equates the channels `c` and `d`, and thus infers the bound 2 for both `c` and `d` even though the ideal bound is 1.

$$1.\texttt{x := c}; \texttt{x := d} \;||\; 2.\texttt{c!}(0) \;||\; 3.\texttt{d!}(0)$$

Hence, the analysis may need to be coupled with a more powerful alias analysis to analyze programs that extensively use channels as values.

7 Related Work

Closely related work is Kobayashi et al.'s type and effect system [3] for inferring the upper bound on the number of pending inputs and outputs on rendezvous channels. There are several differences from our work with theirs. One is that their system relies more on the syntactic structure of the program to determine who is responsible to send and receive capabilities (viewing their effect constraints as capability sends and receives). For instance, if there are multiple reads in a succession then the last read is responsible for receiving all the necessary capabilities. In contrast, our analysis allows more freedom on who can send and receive capabilities, and lets linear programming choose the optimal amount of capabilities to send and receive. For example, in the program below, the optimum buffer space for the channel `c` is 1, which our analysis is able to infer.

$$1.\texttt{c!}(0) \;||\; 2.\texttt{b?}(\texttt{x}); \texttt{a?}(\texttt{x}); \texttt{c!}(1) \;||\; 3.\texttt{c?}(\texttt{y}); \texttt{b!}(0) \;||\; 4.\texttt{a!}(0)$$

But because $\texttt{c!}(1)$ is preceded immediately by $\texttt{a?}(\texttt{x})$, Kobayashi et al.'s system infers the bound 2 instead. Another difference is the use of fractions (i.e., rational arithmetic) that allows our system to have a polynomial time type inference via linear programming. Also, some programs (e.g., the running example) require fractions to infer the optimal buffer bound.

The main technique used in our analysis, passing of fractional capabilities, was used for the purpose of checking determinism of concurrent programs [6]. Fractional capabilities were invented for the purpose of allowing concurrent reads of reference cells [1, 5], and capability calculus was originally proposed for reasoning about resources in sequential programs [2]. In previous applications of fractional capabilities, linear programming was used only to find a satisfying solution to a set of linear inequality constraints, whereas our work makes use of the objective function to find the minimum solution.

8 Conclusions

We have presented a static analysis for inferring the buffer bound of concurrent programs communicating via buffered channels. We have cast the analysis as a capability calculus with fractional capabilities where capabilities can be passed at channel communication point. Our analysis reduces the problem to linear programming and runs in time polynomial in the size of the program.

References

1. J. Boyland. Checking interference with fractional permissions. In *Static Analysis, Tenth International Symposium*, pages 55–72, San Diego, CA, June 2003.
2. K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 262–275, San Antonio, Texas, Jan. 1999.

3. N. Kobayashi, M. Nakade, and A. Yonezawa. Static analysis of communication for asynchronous concurrent programming languages. In *Static Analysis, Second International Symposium*, pages 225–242, Glasgow, Scotland, Sept. 1995.
4. B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, St. Petersburg Beach, Florida, Jan. 1996.
5. T. Terauchi and A. Aiken. Witnessing side-effects. In *10th ACM SIGPLAN International Conference on Functional Programming*, pages 105–115, Tallinn, Estonia, Sept. 2005.
6. T. Terauchi and A. Aiken. A capability calculus for concurrency and determinism. In *Concurrency Theory, 17th International Conference*, volume 4137, pages 218–232, Bonn, Germany, Aug. 2006.

A Proof of Theorem 1

Lemma 2. *Suppose* $\Gamma \vdash e : \tau$, $\Gamma \vdash h$, *and* $(h, e) \Downarrow v$. *Then* $\Gamma \vdash v : \tau$.

Proof. By induction on the type derivation.

Lemma 3. *Suppose* $\Gamma, \Psi_1, \ldots, \Psi_n, \Psi_B \vdash (B, S, p_1)$ *and* $(B, S, p_1) \rightarrow (B', S', p_2)$. *Then there exist* $\Psi'_1, \ldots, \Psi'_n, \Psi'_B$ *such that*

(a) $\Gamma, \Psi'_1, \ldots, \Psi'_n, \Psi'_B \vdash (B', S', p_2)$
(b) Let $\Psi_p = \Psi_B + \sum_{j=1}^{n} \Psi_j$ *and* $\Psi'_p = \Psi'_B + \sum_{j=1}^{n} \Psi'_j$. *Then, for each channel* c, $\Psi'_p(\rho) + \sum_{c' \in HC_{B'}(\rho,\Gamma)} |B'(c')| \leq \Psi_p(\rho) + \sum_{c' \in HC_B(\rho,\Gamma)} |B(c')|$ *where* $\rho = hdl(\Gamma(c))$.

Proof. The proof is by case analysis on $(B, S, p_1) \rightarrow (B', S', p_2)$. We just show the key cases. First, note that (b) can be restated so that the statement is "for each ρ, ..." instead of "for each c, ... where $\rho = hdl(\Gamma(c))$." We use this form as it is more convenient.

Consider the case $(B, S, p_1) \rightarrow (B', S', p_2)$ is an instance of **Write**, that is,

$$\frac{(S(i_j), e_1) \Downarrow c \quad (S(i_j), e_2) \Downarrow v \quad B' = B.write(c, v)}{(B, S, i_j.e_1\,!\,(e_2); s \mid\mid p) \rightarrow (B', S, i_j.s \mid\mid p)}$$

Without loss of generality, let $j = 1$. We have

$$\frac{\Gamma \vdash e_1 : ch(\rho, \tau, \Psi') \qquad \Gamma \vdash e_2 : \tau}{\Gamma, \Psi_1 \vdash e_1\,!\,(e_2) : \Psi_1 - \Psi' - 0[\rho \mapsto 1]}$$

Let $\Psi'_1 = \Psi_1 - \Psi' - 0[\rho \mapsto 1]$. Let $\Psi'_j = \Psi_j$ for $j \neq 1$. Let $\Psi'_B = \sum_{c \in dom(B')} |B'(c)| \times writeSend(\Gamma(c))$. Then we have $\Gamma, \Psi'_1, \ldots, \Psi'_n, \Psi'_B \vdash (B', S, i_j.s \mid\mid p)$. Thus (a) holds.

By Lemma 2, $hdl(\Gamma(c)) = \rho$. Let $\Psi'_p = \Psi'_B + \sum_{j=1}^{n} \Psi'_j$ and $\Psi_p = \Psi_B + \sum_{j=1}^{n} \Psi_j$. Clearly, for $\rho' \neq \rho$,

$$\Psi'_p(\rho') + \sum_{c' \in HC_{B'}(\rho', \Gamma)} |B'(c')| = \Psi_p(\rho') + \sum_{c' \in HC_B(\rho', \Gamma)} |B(c')|$$

Also, because $\Psi'_B + \Psi'_1 = \Psi_B + \Psi_1 - 0[\rho \mapsto 1]$ and $|B'(c)| = |B(c)| + 1$,

$$\Psi'_p(\rho) + \sum_{c' \in HC_{B'}(\rho,\Gamma)} |B'(c')| = \Psi_p(\rho) + \sum_{c' \in HC_B(\rho,\Gamma)} |B(c')|$$

Thus (b) holds.

Consider the case $(B, S, p_1) \to (B', S', p_2)$ is an instance of **Read**, that is,

$$\frac{(S(i_j), e) \Downarrow c \quad (B', v) = B.read(c) \quad S' = S[i_j \mapsto S(i_j)[x \mapsto v]]}{(B, S, i_j.e?(x); s \mid\mid p) \to (B', S', i_j.s \mid\mid p)}$$

Without loss of generality, let $j = 1$. We have

$$\frac{\Gamma \vdash e : ch(\rho, \Gamma(x), \Psi')}{\Gamma, \Psi_1 \vdash e?(x) : \Psi_1 + \Psi' + 0[\rho \mapsto 1]}$$

Let $\Psi'_1 = \Psi_1 + \Psi' + 0[\rho \mapsto 1]$. Let $\Psi'_j = \Psi_j$ for $j \neq 1$. Let $\Psi'_B = \sum_{c \in dom(B')} |B'(c)| \times writeSend(\Gamma(c))$. Then we have $\Gamma, \Psi'_1, \ldots, \Psi'_n, \Psi'_B \vdash (B', S', i_j.s \mid\mid p)$. Thus (a) holds.

By Lemma 2, $hdl(\Gamma(c)) = \rho$. Let $\Psi'_p = \Psi'_B + \sum_{j=1}^{n} \Psi'_j$ and $\Psi_p = \Psi_B + \sum_{j=1}^{n} \Psi_j$. Clearly, for $\rho' \neq \rho$,

$$\Psi'_p(\rho') + \sum_{c' \in HC_{B'}(\rho',\Gamma)} |B'(c')| = \Psi_p(\rho') + \sum_{c' \in HC_B(\rho',\Gamma)} |B(c')|$$

Also, because $\Psi'_B + \Psi'_1 = \Psi_B + \Psi_1 + 0[\rho \mapsto 1]$ and $|B'(c)| = |B(c)| - 1$,

$$\Psi'_p(\rho) + \sum_{c' \in HC_{B'}(\rho,\Gamma)} |B'(c')| = \Psi_p(\rho) + \sum_{c' \in HC_B(\rho,\Gamma)} |B(c')|$$

Thus (b) holds.

Theorem 1. *Suppose $\Gamma, \Psi_1, \ldots, \Psi_n, \Psi_B \vdash (B, S, p)$. Suppose $hdl(\Gamma(c)) = \rho$. Let $\Psi_p = \Psi_B + \sum_{j=1}^{n} \Psi_j$. Then the buffer bound of c in (B, S, p) is within $\Psi_p(\rho) + \sum_{c' \in HC_B(\rho,\Gamma)} |B(c')|$.*

Proof. Straightforward from Lemma 3.