University of Glasgow | School of Computing Science

Move Analysis: An Interprocedural Data Flow Analysis for Efficient Message Passing

Craig McLaughlin

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — March 28, 2014

Abstract

The Insense programming language is a domain-specific language for programming wireless sensor networks supporting both an efficient mode of communication between devices and a scalable concurrency model. The aim of this work is to extend Insense to improve the random access memory utilisation on devices such as the T-Mote Sky during network communication. We present an interprocedural data flow analysis to enforce soundness of a language extension used to improve the efficiency of the message passing mechanism within the Insense programming language. We provide empirical results to show the improvements obtained from utilising the extension, including overall memory consumption, less severe external fragmentation, and larger free blocks on memory-constrained devices.

Acknowledgements

Thanks to my co-supervisors, Professor Joseph Sventek and Paul Harvey, for many a fruitful discussion throughout this project.

I would also like to thank Callum Cameron for his, greatly appreciated, assistance during the evaluation phase of this work.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: _____ Signature: _____

Contents

Chapter 1

Introduction

There are many approaches to programming wireless sensor networks [MP11]. Typically, the devices within such a network are small with random access memory, and program store memory on the order of tens of kilobytes. The nature of the network requires frequent communication between devices mandating the programming language provide an efficient mechanism for concurrency. The space limitations constrain the amount of data that can be buffered during communication.

Insense [Dea+08] is a domain-specific language for programming wireless sensor networks, among other devices, supporting both an efficient mode of communication between devices and a scalable concurrency model. The aim of this work is to extend Insense to improve the random access memory utilisation on devices such as the T-Mote Sky during network communication.

1.1 Insense

Insense is an actor-based language aimed at easing the programmability of wireless sensor nodes for domain experts that are non-specialist programmers. The primary facilities offered by Insense include components (also known as actors), which are stateful objects that maintain complete isolation from the outside environment, and represent a concurrent thread of execution; typed channels, the communication mechanism used in Insense to send information between components using blocking rendezvous semantics [1]; and interfaces, a list of channels that are provided by components implementing the interface i.e. what type of information they can send, and what they can receive.

Similar to the idea of classes in many object-oriented programming languages, a component can have one or more constructors which can be passed arguments. Every component has a behaviour clause, a section of code that is executed, looping indefinitely or until a **stop** statement is reached, once the component is started. The behaviour clause of a component represents the thread which animates the component. Inside a component, the programmer can specify variables that are private to the component and inaccessible to the outside world. Therefore, the only way to pass data between components is to use the channel communication mechanism.

Channels are defined as being either **in**, or **out**, depending on whether a component will receive information from the channel, or send information over the channel, respectively. The type of data passed through a channel is specified in the channel declaration. Insense provides **send** and **receive** primitives to facilitate passing of data

[1] In other words, sending components will block until the corresponding receiver accepts the message, and receiving components will block until there is a message sent from the corresponding sender. This communication scheme is also known as synchronous communication.

across an **in/out** channel pair. The **send** operation performs a deep-copy of the data to prevent state from being shared with the sending component.

Listing 1.1 provides an example of an Insense application. The main program creates a sensor component, then a logger component, and then connects the **out** channel of the sensor with the **in** channel of the logger. Thus, the system can now log temperature readings retrieved from the environment by the sensor.

In order to communicate a message is sent across the channel, in this case a temperature reading. The reading is then logged by the logger. After the temperature has been sensed, the sensor goes off to do some long computational task unrelated to the sensing of temperatures. The channel performs a deep-copy of the memory passed to it since a component's state is private, and therefore cannot be sent directly. Thus, the memory occupied by the sensedTemp reference variable is unused in the Sensor component after the **send** operation. Heap memory in Insense is managed by reference-counted garbage collection, and because the sensedTemp reference is still in scope, the memory allocated to it cannot be used for other resources. This represents a problem if the data to be sent is relatively large with respect to the random access memory capacity of the device.

1.2 Memory Optimisation

So what can be done to solve such memory inefficiencies? It would appear the problem lies in a component no longer needing a resource after it has sent it across a channel to a receiving component. However, since Insense does not allow the programmer direct access to memory, it is not possible for the programmer to deallocate the memory herself. Indeed, this approach would be very error prone. Instead, the project brief is to define a new "memory type" for the Insense programming language. The new type would allow references to be sent across a channel between communicating components without the need for an expensive deep-copy operation, freeing up space on memory-constrained devices, and preventing the programmer from accessing memory no longer available from within the sending component. Ideally, the memory type extension would not require alteration of the type inferred nature of Insense since such a change may make it more difficult for non-experts to use.

The approach taken in this work is to extend the Insense programming language to provide a new annotation (**mov**) to heap allocation points. The Insense compiler is extended to track the movability of heap-allocated objects using *data flow analysis*, a technique commonly used in compilers to discern the properties of a program (usually with, but not limited to, the aim of performing transformations on the internal representation of the program that will optimise the program's size or execution time characteristics). Before we discuss the contributions of this work in detail, we must first introduce concepts from the literature used herein.

Listing 1.1: Example Insense application

```
type TempReading is struct(integer epoch ; real temp)
type ISensor is interface(out TempReading temp)
type ILogger is interface(in TempReading temp)

component Sensor presents ISensor {
    sensedTemp = new TempReading(0, 15.0);

    constructor() {
    }

    proc sense() {
        // ...
    }

    proc doSomeUnrelatedLongComputation() {
        // ...
    }

    behaviour {
        sense();
        send sensedTemp on temp;
        doSomeUnrelatedLongComputation();
        stop;
    }
}

component Logger presents ILogger {
    constructor() {}

    behaviour {
        receive tr from temp;
        log(tr);
    }

    proc log(TempReading tr) {
        // do some logging...
    }
}

sensor = new Sensor();
logger = new Logger();
connect sensor.temp to logger.temp;
```

Chapter 2

Background

This chapter presents the relevant theory and context for the project.

2.1 Compiler Theory

This section presents the relevant theoretical background in compilers necessary to understand the subsequent material in this work. We describe the common data structures used within a compiler, the framework used to analyse programs, and the algorithms and techniques used to facilitate the analysis of program properties.

2.1.1 Preliminaries

We begin with some preliminary defintions of the foundational mathematics behind the compiler analysis techniques that will be introduced later. The following definitions are taken, or adapted, from [Kil73; KU76; KSK09].

Defintion 2.1.1. [Kil73] A finite meet semilattice, L, is a set with a binary meet operator, $\sqcap$, with the following properties for all $x, y, z \in L$:

$$\begin{aligned} x \sqcap x &= x && \text{(idempotence)} \\ x \sqcap y &= y \sqcap x && \text{(commutativity)} \\ (x \sqcap y) \sqcap z &= x \sqcap (y \sqcap z) && \text{(associativity)} \end{aligned}$$

Defintion 2.1.2. [Kil73] [KU76] For a finite meet semilattice L, a partial ordering is defined for all $x, y \in L$ such that

$$\begin{aligned} x \leq y &\Rightarrow x \sqcap y = x \\ x < y &\Rightarrow x \leq y \text{ and } x \neq y \\ x \geq y &\Rightarrow x \sqcap y = y \end{aligned}$$

Let L be a finite meet semilattice. The *least* element, denoted $\bot$, of L, is such that $\forall x \in L$, $x \sqcap \bot = \bot$. A sequence, $x_1, x_2, \ldots, x_n$ of elements of L, is called a *chain* if for $1 \leq i < n$ we have $x_i > x_{i+1}$. L is *bounded* if for all $x \in L$ there exists a constant b_x, signalling a bound on the length of any chain beginning with x. We can define $\prod_{x \in S} x$ where $S = \{x_1, x_2, \ldots\}$, to be $\lim_{n \to \infty} \prod_{1 \leq i \leq n} x_i$, for any bounded meet semilattice L.

A *control flow graph* (CFG) [All70] represents a procedure in a program as a directed graph, $G = (N, E)$, where the nodes, N, contain the procedure's instructions, and the edges, E, represent control flow through the procedure. By control flow, we mean the constructs of the source language that determine whether or not a section of code is executed. Traditional control flow constructs are the if, while, and for statements. The CFG, by convention, has a unique entry node, n_{entry}, and a unique exit node, n_{exit}.

The nodes in a CFG represent *basic blocks* as a sequence of instructions in which the only branching code occurring throughout the entire block is located at the end of the block. In other words, if a program enters a block during execution, then all of the instructions inside the block are executed. Each block has zero or more successor blocks, and zero or more predecessor blocks represented by the directed edges of the CFG. Let $succs(n)$ and $preds(n)$ denote the set of (immediate) successors and predecessors respectively, of node n. That is, there is an edge $(p, n) \in E$ for all $p \in preds(n)$, and an edge $(n, s) \in E$ for all $s \in succs(n)$. Let $paths(n)$ denote the set of all paths from n_{entry} to n where a path from n_{entry} to n is a sequence of edges, $(n_{entry}, m_1), (m_1, m_2), \ldots, (m_k, n)$ where $m_1, m_2, \ldots, m_k \in N$, in E.

Defintion 2.1.3. Let $G = (N, E)$ be an arbitrary CFG. We can construct a *reverse post-ordering* of the blocks in G by performing a topological sort on G which assigns a reverse post-order number to the blocks in G (see Algorithm 1, $rpon$ is a globally available variable). For any basic block $n \in N$, let $rpon(n)$ return the reverse post-order number for n. Then after performing the topological sort on G, we have,

- A *forward edge* in E is any $(p, q) \in E$ such that $rpon(p) < rpon(q)$,
- A *back edge* in E is any $(p, q) \in E$ such that $rpon(p) \geq rpon(q)$.

Algorithm 1: Topological sorting of CFG nodes using depth-first search to produce reverse post-ordering.

```
1  void rpo_cfg(CFG G = (N, E))
2  begin
3      rpon ← |N|
4      for n ∈ N do
5          if n is unvisited then
6              dfs_topsort(n)
7  void dfs_topsort(BasicBlock n)
8  begin
9      mark n as visited
10     for s ∈ succs(n) do
11         if s is unvisited then
12             dfs_topsort(s)
13     rpon(n) ← rpon
14     rpon ← rpon − 1
```

The instructions inside a basic block are some intermediate representation of the program such as *three-address code* (also known as *quadruple code*), where each instruction contains at most three addresses and an operator e.g., $x = y \; op \; z$, where x, y, and z are storage locations (representing abstract or physical memory addresses), and op is some binary operator. The exact constructs defined in the intermediate representation will depend on its level of abstraction, and correlation to the source and target languages. Muchnick [Muc97] describes various levels of abstraction for intermediate languages. The design of the intermediate representation used for Insense programs is described in Section 4.2.

A *call graph* (CG) [Ryd79], or call multigraph [Cal+90], is a directed graph, $G = (N, E)$, where the nodes of the graph, N, represent procedures in the program, and the edges of the graph, E, represent calls from the

source node to the destination node. It is a useful representation of the relationship between caller and callee, and can serve as an information repository on procedures during an analysis (see Section 4.4.2.1).

The *supergraph* [Mye81] representation of a program connects the CFGs of callers and callees, using *interprocedural edges*. The supergraph, $G^* = (N^*, E)$, can be constructed as follows:

- Let $\mathbb{P}$ be the set of procedures in the program,
- Represent each procedure call as a single-instruction block (a call is a kind of branching instruction so this representation does not depart from the single-entry/single-exit notion of a basic block),
- Represent each procedure p by a CFG, $G_p = (N_p, E_p)$ with $p_{n_{entry}}$ and $p_{n_{exit}}$ as its unique entry and exit nodes, respectively,
- For each procedure call site i, calling p, in a basic block n, replace the basic block n with two empty basic blocks c_i (called a *call node*), and r_i (called a *return node*),
- Add the intraprocedural edges, $\forall m \in preds(n)$, (m, c_i), and $\forall s \in succs(n)$, (r_i, s), to E_p,
- Add the interprocedural edges, $(c_i, p_{n_{entry}})$ (called a *call edge*), and $(p_{n_{exit}}, r_i)$ (called a *return edge*) to E,
- Set $N^* = \bigcup_{p \in \mathbb{P}} N_p$,
- Set $E = E \cup \bigcup_{p \in \mathbb{P}} E_p$.

The supergraph can be considered to have a unique entry and exit point, similar to a CFG, denoted by n_{entry} and n_{exit}, respectively. An *interprocedurally valid path* is a path in G^* from n_{entry} to a node $n \in N^*$ consisting of a sequence of edges in E such that:

- The path consists of intraprocedural edges only, or
- Any matching pair of interprocedural edges, call edge and return edge, in the path are separated by an interprocedurally valid path.

We denote the set of all interprocedurally valid paths in G^* from n_{entry} to a node $n \in N^*$ as $paths_{G^*}(n)$. Any path $p \notin paths_{G^*}(n)$ is called an interprocedurally invalid path.

2.1.2 Data Flow Analysis

Data flow analysis is the tracking of data within a program, typically program variables and their values, in order to capture properties about the program. The intention is to use the details obtained to optimise the program using some metric (running time, memory consumption, etc). The properties in which we are interested are referred to as the *data flow values*. Data flow analysis of a program is separated into two phases, *intraprocedural* and *interprocedural*.

2.1.2.1 Intraprocedural Analysis

Intraprocedural analysis considers the contents of each procedure in isolation, representing them as CFGs. Global data flow analysis is described by Kildall [Kil73], and Kam and Ullman [KU76] in the context of the control flow graph. A meet semilattice is used to encode the data flow values of interest. In this formulation, the elements of the semilattice are typically subsets of the program variables, or some property thereof. To encode the effect of basic blocks on the data flow values we shall introduce the concept of *admissible flow functions* [KU76]; associated with each basic block n is a function $f_n : L \mapsto L$, which encodes the effect of the instructions, contained within the block, on the data flow values provided. These flow functions can be used to define simultaneous *data flow equations* used to compute the data flow values valid on entry (In_n), and exit (Out_n), of each basic block in a CFG:

$$\begin{aligned} In_n &= \begin{cases} BI & \text{if } n \text{ is } n_{entry} \\ \bigsqcap\limits_{p \in preds(n)} Out_p & \text{otherwise} \end{cases} \\ Out_n &= f_n(In_n) \end{aligned} \tag{2.1}$$

where BI is the data flow information available upon entry to the CFG, for example, data flow information associated with global variables. Whilst we said intraprocedural analysis considers each procedure in isolation, it will be useful to state these equations using some non-empty data flow value as the initialisation for n_{entry} to make extending the model to the interprocedural setting easier. To consider intraprocedural analysis in isolation we could simply set $BI = \bot$. We perform a meet over the Out sets of the predecessor blocks to obtain the data flow information on entry to block n.

For a bounded meet semilattice L, Kam and Ullman [KU76] defined the necessary properties that a set of functions, F, mapping L to L, must satisfy to be an admissible set of flow functions for L:

$$\begin{aligned} \forall f \in F, \forall x, y \in L : f(x \sqcap y) = f(x) \sqcap f(y) & \quad \text{(distributivity)} \\ \exists id \in F, \forall x \in L : id(x) = x & \quad \text{(identity function)} \\ \forall f, g \in F \Rightarrow (f \circ g) \in F & \quad \text{(closure under function composition)} \\ \forall x \in L, \exists H \subseteq F : x = \bigsqcap_{h \in H} h(BI) & \quad \text{(existential meet)} \end{aligned}$$

where $\sqcap$ is the meet operator for L. The existential meet property ensures that the elements of L are only those data flow values which can been computed from a finite meet of the admissible flow functions from F applied to the initialising data flow value for the CFG.

We define a distributive data flow framework as a tuple, $D = (L, \sqcap, F)$, where L is a bounded meet semilattice, $\sqcap$ is the meet operator of the lattice, and F is a set of admissible flow functions for L. A framework may be instantiated with an instance, $I = (G, M)$, where G is the particular CFG under consideration, and M is a function mapping basic blocks within G to flow functions in F.

Algorithm 2 computes the data flow sets In_n for each basic block n in the input CFG. The algorithm is from the description provided by Khedker, Sanyal, and Karkare [KSK09, p. 90]. The input is an instance for a distributive data flow framework, with f_n denoting the function, retrieved from the basic block to flow function mapping M, for block n. The blocks in G are numbered in reverse post-order. Algorithm 2 computes what is termed the *maximal fixed point* (MFP) assignment.

Let $d(G)$, the loop-connectedness [KU76] of a CFG G, be the largest number of back edges in any acyclic path of G. We have the following theorem:

Theorem 2.1.4. *[KU76] [KSK09] Let $D = (L, \sqcap, F)$ be a distributive data flow framework. Then Algorithm 2 halts after at most $d(G) + 3$ iterations for every instance $I = (G, M)$ of D and every reverse post-ordering*

Algorithm 2: General intraprocedural data flow algorithm.

```
1  void dfa(Instance (G = (N, E), M))
2  begin
3      In_0 = BI
4      for i = 1 to |N| − 1 do
5          In_i = ⊓_{p∈preds(i)∧rpon(p)<rpon(i)} In_p
6      change ← true
7      while change do
8          change ← false
9          for i = 1 to |N| − 1 do
10             temp = ⊓_{p∈preds(i)} In_p
11             if temp ≠ In_i then
12                 In_i ← temp
13                 change ← true
```

definable for $G = (N, E)$, if and only if D satisfies:

$$\forall f, g \in F, \forall x, BI \in L : (f \circ g)(BI) \geq g(BI) \sqcap f(x) \sqcap x \tag{2.2}$$

Theorem 2.1.4 is the famous "rapid" condition for distributive data flow frameworks.

Lemma 2.1.5. *[KSK09] Bit vector frameworks are rapid.*

A data flow analysis is *flow-sensitive* if it takes account of the control flow. It is *flow-insensitive* if it computes the effect of each block irrespective of the control flow.

Intraprocedural analysis does not handle the effect of procedure calls within a procedure's CFG, and the effect of the procedure called is either ignored, or approximated. Thus, the data flow information obtained from intraprocedural analysis is imprecise due to the approximation of procedure calls. To increase the precision of the data flow analysis, we must take account of procedure calls by performing interprocedural analysis.

2.1.2.2 Interprocedural Analysis

Interprocedural analysis considers the effects of procedure calls on data flow values. That is, if at program point n there is a procedure call, interprocedural analysis analyses the called procedure to compute the effect of the procedure's statements on the data flow values in In_n. Two main approaches have been described in the literature, the functional approach and the call-strings approach [SP78; SP81].

The functional approach maintains the separation of the control flow graphs of each procedure from each other. The effect of a function on data flow values incoming to a call of the function are encoded by a *summary flow function*, a composition of the effects of the individual instructions within the function definition. Whilst this approach is efficient since it analyses the effect of functions only once, the call-strings approach is adopted in this work because it results from a simple extension of the intraprocedural case, and can be understood intuitively as a simulation of actual program execution (the call stack). We shall now focus entirely on the call-strings approach for the remainder of this work.

In contrast to the functional approach, the call-string approach operates on the supergraph representation of a program. The key concept in the call-strings approach for interprocedural data flow analysis is to maintain a *token* stack, where each token represents an uncompleted call to a procedure i.e. a procedure call which has not yet returned. The tokens on the stack are known as the *call string*, or the *calling context*. A data flow analysis is *context-sensitive* if it distinguishes between different calling contexts, meaning data flow information can be propagated back to the specific call site under analysis. Conversely, a *context-insensitive* analysis merges the information from all calling contexts resulting in potentially imprecise information for callers due to data flow information being generated for interprocedurally invalid paths.

To take account of interprocedural effects we must somehow keep a record of the data flow values with respect to the token stack. The solution is to use *qualified data flow values*. The data flow equations described in Section 2.1.2.1 are extended to operate on pairs of the form $< \gamma, x >$, where γ is the calling context (a sequence of call sites concatenated together representing the token stack), and x represents the data flow information of the original problem in the specified context. Let λ denote the empty calling context. We define the qualified data flow values at each $n \in N^*$, QIn_n and $QOut_n$, as:

$$\begin{aligned} QIn_n &= \begin{cases} < \lambda, BI > & \text{if } n \text{ is } n_{entry} \\ \biguplus_{p \in preds(n)} Out_p & \text{otherwise} \end{cases} \\ QOut_n &= \begin{cases} \{< \gamma \circ c_i, x > \mid < \gamma, x > \in In_n\} & \text{if } n \text{ is } c_i \\ \{< \gamma, x > \mid < \gamma \circ c_i, x > \in In_n\} & \text{if } n \text{ is } r_i \\ \{< \gamma, f_n(x) > \mid < \gamma, x > \in In_n\} & \text{otherwise} \end{cases} \end{aligned} \tag{2.3}$$

where $\circ$ concatentates the call node onto the calling context, f_n is the admissible flow function for node n, and $\biguplus$ is defined as [KSK09]:

$$\begin{aligned} X \uplus Y = &\{< \gamma, x \sqcap y > \mid < \gamma, x > \in X, < \gamma, y > \in Y\} \cup \\ &\{< \gamma, x > \mid < \gamma, x > \in X, \forall z \in L, < \gamma, z > \notin Y\} \cup \\ &\{< \gamma, y > \mid < \gamma, y > \in Y, \forall z \in L, < \gamma, z > \notin X\} \end{aligned}$$

If during the analysis we encounter a call node, it is appended to the context, and data flow information is propagated along the call edge to the entry node of the CFG representing the callee. When a return node is encountered, the corresponding call node is removed from the context, and the data flow information is propagated along the return edge to return node of the CFG representing the caller. For all other nodes in the supergraph the flow functions from the intraprocedural data flow framework are applied to the data flow value x and the context is unchanged.

Equations 2.3 are used to compute the data flow values, In_n and Out_n, for each node $n \in N^*$ by performing a meet over the sets of pairs:

$$\begin{aligned} In_n &= \bigsqcap_{< \gamma, x > \in QIn_n} x \\ Out_n &= \bigsqcap_{< \gamma, x > \in QOut_n} x \end{aligned} \tag{2.4}$$

For programming languages without recursion the maximum length of the calling context is bounded by the depth of the call graph. Therefore, if the original data flow problem terminates so too does the interprocedural extension of the problem.

2.2 Literature Review

This section presents a review of the relevant literature for the project. The focus is compiler analysis techniques for reasoning about program properties, and related efforts in programming language research.

2.2.1 Movability

Movability is similar in nature to other concepts within programming language research. We can think of sending a block of memory across a communications channel in Insense as an operation on the type of the memory. If the memory has type **mov** then the operation may only be applied once, otherwise it is valid any number of times. Additionally, once sent, a **mov** typed memory object may not be used again. Thus, the set of applicable operations on a particular type changes depending on the context in which the type is used. The notion of typestate [SY86] captures this general concept of contextualising operations, of which movability is a special case. Essentially, **mov** enforces the notion of a destructive send operation as described in the paper. The approach described by Strom and Yemini performs a kind of intraprocedural data flow analysis to validate a program, but requires annotations to specify the effect of a procedure call on the typestate of a variable, in contrast to our approach where we perform fully context-sensitive interprocedural analysis in addition to intraprocedural analysis. The advantage of interprocedural analysis over annotations is the compiler handles the entire checking process whereas manually writing annotations can be a tedious and error-prone task for the programmer. Strom and Yemini's algorithm for tracking typestate performs flow-insensitive aliasing by coalescing aliased variables to a single identifier. Our approach is more precise, using flow-sensitive may-alias analysis to allow a variable's alias set to change over the execution of the program.

Hogg [Hog91] describes the use of *bridges* to encapsulate *islands* of state within object-based languages in the presence of aliasing. Conceptually, an Insense component is a stricter form of a bridge, since the state inside a component will never escape its scope. Destructive reads are similar to the movability property. Our approach differs in syntactic cost; the compiler tracks movables throughout the program from a single annotation at the object allocation point (**new**), whereas (in addition to object allocation points) the "access mode" of parameters and function results must be specified in [Hog91].

Rust [Moz14] has a similar memory and concurrency model to Insense. A *task* in Rust is similar to a component with defined communication channels. In Rust, communication is performed using *pipes*; a *channel* is a sending endpoint of a pipe, a *port* is a receiving endpoint for a pipe. The notion of channels and pipes are equivalent to **in** and **out** channels in Insense, respectively. Tasks cannot share data with each other and must transfer ownership using a global *exchange heap*. The *Send trait* acts to communicate between tasks such that the memory can be transferred ensuring that it is no longer used by the sender after being sent. The Send trait allows only *owned* boxes to be sent between tasks, where an owned box is an object that has a single pointer (owning pointer) to it. A *managed* box is an object that can have any number of pointers (managed pointers) to it. The heap can be viewed as split into regions for owned and managed boxes, respectively.

The notion of *borrowing* is provided in Rust by *borrowed pointers*, to which managed or owning pointers can be assigned using an automatic pointer conversion operation provided by the language. For example, an owning pointer can be borrowed by passing it as an argument to a function accepting a borrowed pointer. During the execution of the function (known as the *lifetime* of the borrowed pointer), the owning pointer cannot be used since the object is on *loan* to the function; the function is known as the *borrower*. When the borrower returns, the owning pointer may be used again. Further, a borrower cannot send the object over a communication channel to another task; the object may only be sent from an owning pointer. These rules for move semantics require the programmer to know about lifetimes and pointer conversion operations. Non-specialist programers may find it difficult to grasp such concepts easily.

The **mov** property is an almagmation of the managed and owning pointers in Rust- allowing multiple references to the object within a component (analogous to multiple references within a task)- yet still maintains the task-level ownership analogous to an owning pointer in Rust. Additionally, there is no need for the programmer to understand lifetimes or pointers since these concepts are not made explicit in the move semantics defined for Insense (see Section 3.1). Instead, the programmer simply annotates heap allocation points with **mov** and the compiler tracks references to all such objects. No annotations are required to pass such references as arguments

to functions or return them as results, the movability of references is tracked entirely by the compiler analysis (see Section 3.2).

Ownership types [CPN98] were developed for providing strict, static object encapsulation for object-based languages. Insense provides the semantics of *rep* for all components by definition, and a component is the *owner* of its entire state. "Ownership transfer" is conceptually the same as the movability property, allowing objects to "jump" across the *articulation point* represented by the component. In the object graph described by Clarke, Potter, and Noble [CPN98] there is no dual of the Insense channel mechanism but we can think of these as gateways or bridges (as in [Hog91]) to allow an object to move across the boundary. It is also worth noting that alias protection is reserved with movability since only one component will have any aliases to an object at one time, though that component can have any number of aliases of its own.

Naden et al. [Nad+12] present a type system that allows transference of object ownership by using a "permission" based mechanism. The system provides a simple mechanism for procedures to borrow, or even consume, an object through changing the permission attribute; a more powerful and flexible notion than linear types [Wad90], where conversion to a linear (corresponds to "unique" in [Nad+12]) type is stricter. The system could easily provide higher-level abstractions on top of the defined permission semantics to provide a form of the movability property. Indeed, a **mov** type acts in much the same way as an object initially set with a "shared" permission. Send and receive primitives could be defined as functions which change the permission of the provided object to "none". The system also gives the programmer tight control of aliasing through the permission system. The programmer has to explicitly manage the permissions, and have detailed knowledge of language theory to understand the effect of permissions on aliasing, which could detract from the task of developing the application, obscuring the workings of the algorithm to be implemented with type theory concepts. Our approach tries to minimise the burden placed on the programmer, requiring very few changes to a source program to enable the extension, and leaving the analysis to perform the movability tracking throughout the program and the management of aliases.

2.2.2 Error Detection Using Data Flow Analysis

Static analysis techniques have been employed to determine whether programs contain errors. Fosdick and Osterweil [FO76] describe an approach for detecting errors in FORTRAN programs using static data flow analysis techniques. Their approach utilises global data flow analysis techniques from Hecht and Ullman [HU73]. In general, the technique is flow-sensitive but not path-sensitive, where path-sensitive means generating information based on a single path within a control flow graph rather than merging information at nodes with multiple predecessors. However, Fosdick and Osterweil suggest extensions to the technique to treat potentially erroneous paths as path-sensitive. Aliasing between formal parameters is not considered, thus context sensitivity is limited. Additionally, as in this work, recursion is not handled.

The Broadway compiler system [GBL02] used annontations to provide safety guarantees from the programmer to the runtime including those to ensure there are no format string vulnerabilities or deadlocks. The annotations define their own data flow analysis lattice, and are utilised by a "scalable aggressive" compiler analysis to detect programming errors such as privacy issues, and security vulnerabilities. The approach is similar to this work in that a context-sensitive and flow-sensitive interprocedural analysis (call-strings approach) is used to provide precise data flow information. The use of annotations is more burdensome for the programmer yet allows library routines that have been pre-compiled to be integrated into a new software package provided the interface is kept up-to-date. Guyer, Berger, and Lin must bound the calling context since C allows recursion, resulting in imprecision for large programs; our method does not suffer from such imprecision due to the restriction on recursion imposed by Insense (see Section 3.1).

Chapter 3

Theoretics of Move Analysis

This chapter summarises the semantics for defining movable memory in Insense, and develops the theoretical basis for the implementation. We introduce the steps necessary to perform move analysis on an Insense program. Namely, the type-checking that ensures correct usage of the language extensions imposed by movability (in particular, the **copy** and **mov** keywords), and the data flow equations used to compute, for each program point, the set of movable objects that *cannot* be *used*. A *use* is defined to be any reference to a variable that does not re-define that variable e.g. $x = x$, is a use of x, denoted $use(x)$, since the value of x is loaded before the assignment.

3.1 Insense Move Semantics

The aim of this section is to enumerate the semantics of the movability property. It is important to first highlight some restrictions imposed on the programmer by Insense. Firstly, the programmer may not define truly (mutually) recursive structures. The programmer may implicitly define such a structure using the *any* type, which signifies a generic object, but the object must be surrounded with the "any()" cast expression. Additionally, the programmer cannot dereference an *any* type, and must use a **project** statement (see Figure 3.1) to access the fields of an object assigned to an *any* reference. Secondly, it is not possible to define (mutually) recursive procedures (or functions) in Insense. As stated in Section 2.1.2.2, this restriction guarantees that if the intraprocedural version of the data flow problem terminates so too does the interprocedural extension of the problem. Thus, we can focus our termination concerns on the intraprocedural setting alone.

Conceptually, we can view movability as a property of the memory we are referencing. In other words, we can view the heap of an Insense program as two separate spaces (Figure 3.1). Each component in Figure 3.1 has its own local heap space, and shares the movable memory heap (also known as *exchange heap* [Hun+07; HL07]) space with other components in an application. Note this model is purely conceptual, and there is no actual partitioning of the heap.

Consider the Insense program in Listing 3.2. The **mov** keyword instructs the compiler to create the heap object in the movable heap memory space. In the listing, x is a reference which references a movable heap object, z (w, and y) reference "local" (non-movable) heap object(s). The assignment $y := x$, modifies the reference y to refer to the heap object referenced by x. In other words, it is much the same as a pointer assignment in C. Similarly, the assignment $x := z$, performs a pointer assignment on x such that it will now refer to the same Bar object as z. At point (1), x is sent across the channel, and since x refers to an object on the component local heap, the object will be deep-copied and the copy sent, the same is true for sending w, and z. However, when y is sent across the channel a copy is not made, and a pointer to the object referenced by y will be sent across the

Listing 3.1: Example of projecting an any typed variable.

```
type IAny is interface()
type Complex is struct(real a; real b)

component AnyTest presents IAny {
    constructor() {
    }
    behaviour {
        a = any(new Complex(5.0, 3.0));

        // Output: Complex a = 5.0 + 3.0i
        project a as value onto
            Complex : {
                printString("Complex␣a␣=␣");
                printReal(value.a);
                printString("␣+␣");
                printReal(value.b);
                printString("i\n");
            }
            real : {
                printString("real\n");
            }
            default : {
                printString("neither!\n");
            }
        stop;
    }
}

// Insense main
s = new AnyTest();
```

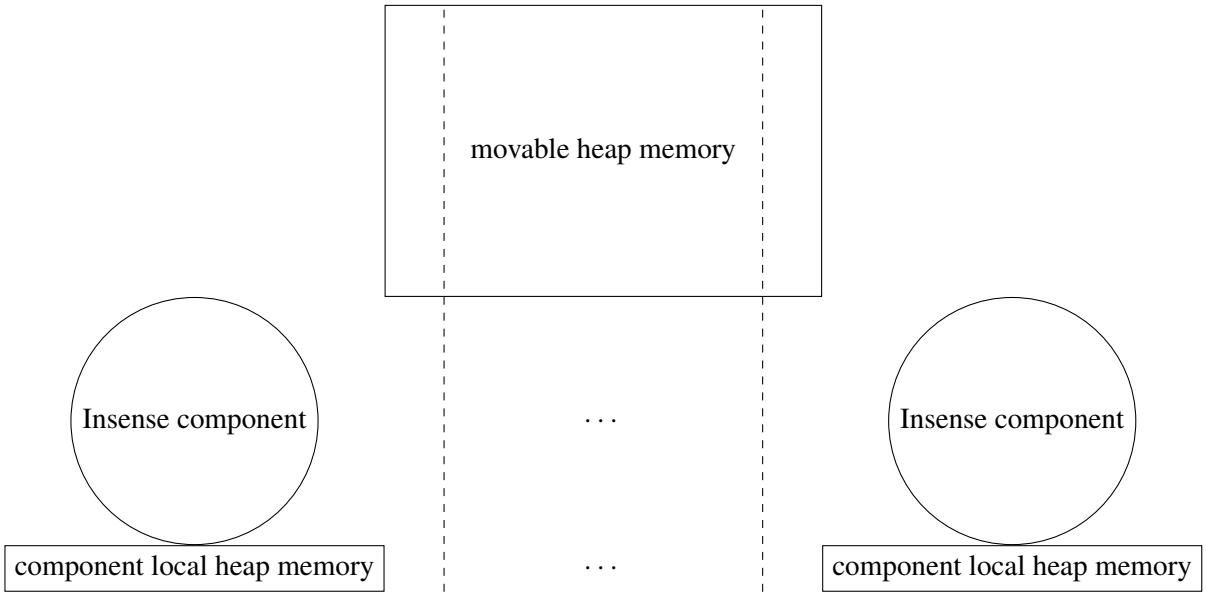


Figure 3.1: Conceptual view of heap memory in Insense

Listing 3.2: Associating the movability with the memory object

```
component Sensor presents ISensor {

    behaviour {
        x = mov new Blah();
        w = new Foo();
        y = new Foo();
        z = new Bar();
        y := x; // y now references a movable Blah object
        x := z; // x now references a non-movable Bar object
        send x on refBarOut; // (1) deep-copy the Bar object referenced by x
        send y on refBlahOut; // (2) move the Blah object referenced by y
                              // and invalidate all references to it.
    }
}
```

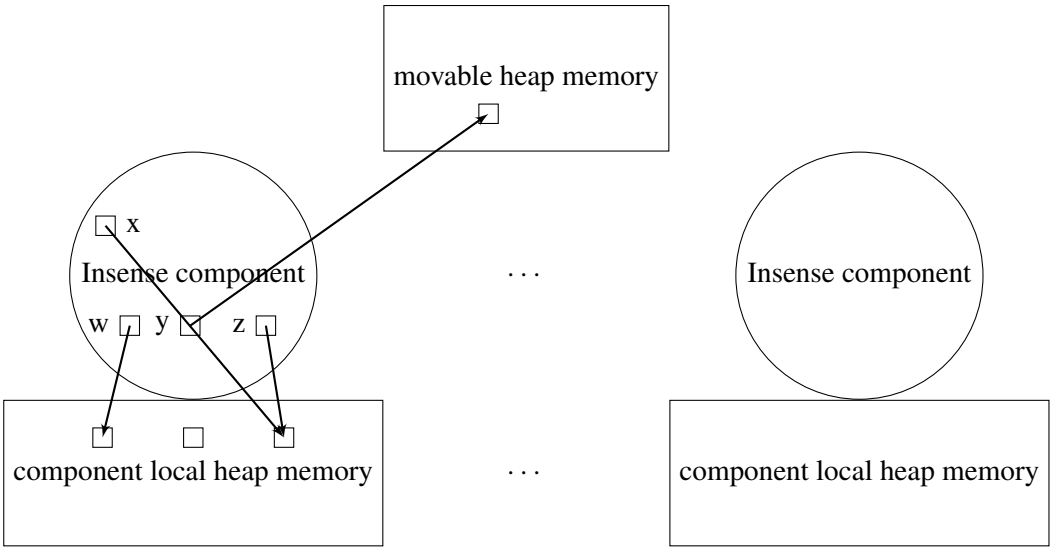


Figure 3.2: Result of assignments before send operation on x in Listing 3.2

channel. We say that y (and all other references which refer to the $Blah$ object sent) has been *invalidated* by the send operation. A use of y after point (2) in Listing 3.2 would be detected at compile-time by the analysis, and the compiler would generate an error since the memory y references has been moved to the receiving component, and is no longer accessible to the Sensor component.

In the example, a behaviour clause has been used, and it should be noted that a behaviour clause loops indefinitely or until a **stop** is encountered. Section 3.1.3 gives more details on the use of **stop** in the presence of movability. The key point is, in every iteration of the behaviour loop, a new movable heap object is being allocated and assigned to x so there are no accesses to invalid memory.

Figure 3.2 shows the situation before (1) in Listing 3.2. Note that in reality both x, y, w, and z will themselves be in the local heap memory area associated with the owning component but to simplify the example the Figure places them inside the Insense component.

Listing 3.3: struct examples demonstrating movable semantics.

```
type simpleStruct is struct(integer i);
type complexStruct is struct(integer i ; simpleStruct s);

component Sensor presents ISensor {
    simpleA = new simpleStruct(1);
    simpleB = mov new simpleStruct(10);

    complexA = new complexStruct(1, simpleA);
    complexB = new complexStruct(10, simpleB);
    complexC = mov new complexStruct(10, simpleA);
    complexD = mov new complexStruct(10, simpleB);

    behaviour {
        send simpleA on chan; // deep copy
        send simpleB on chan; // send reference and invalidate simpleB

        send complexA on chan; // deep copy
        send complexB on chan; // deep copy complexB, but use the reference
                               // of simpleB. Invalidate simpleB
        send complexC on chan; // pass the reference of complexC, and
                               // deep copy simpleA. Invalidate complexC
        send complexD on chan; // just pass the reference of complexD.
                               // invalidate complexD and simpleB
    }
}
```

3.1.1 Semantics of Composite Data Types

Insense provides support for composite data types in the form of arrays and structures. The semantics for movability with regard to composite data types are described in this section.

3.1.1.1 Structure Types

A structure type in Insense contains a number of fields of the Insense primitive type, or a reference. A structure or array variable is considered a reference type in Insense, allowing nesting of structures.

The semantics for structure data types in Insense is to recursively apply the move semantics for references to heap memory e.g. a structure denoted movable will have all primitive fields moved, and the references within the structure will have their movability checked to determine whether to deep-copy or move the reference. Listing 3.3 highlights a number of the possible combinations of movable-nonmovable heap objects within structures. The first two **send** cases follow the same rules as described above. Sending $complexA$ will require (1) copying the reference held to $simpleA$, (2) creating a copy of $complexA$, and (3) assigning the copy of $simpleA$ to field s of the new copy of $complexA$. At the other end of the movable-nonmovable continuum, $complexD$ requires no copying, and all references can be sent freely. The $complexC$ case is illustrated in Figure 3.3, and involves a deep-copy of $simpleA$, followed by a pointer update to set $complexC.s$ to reference the new copy.

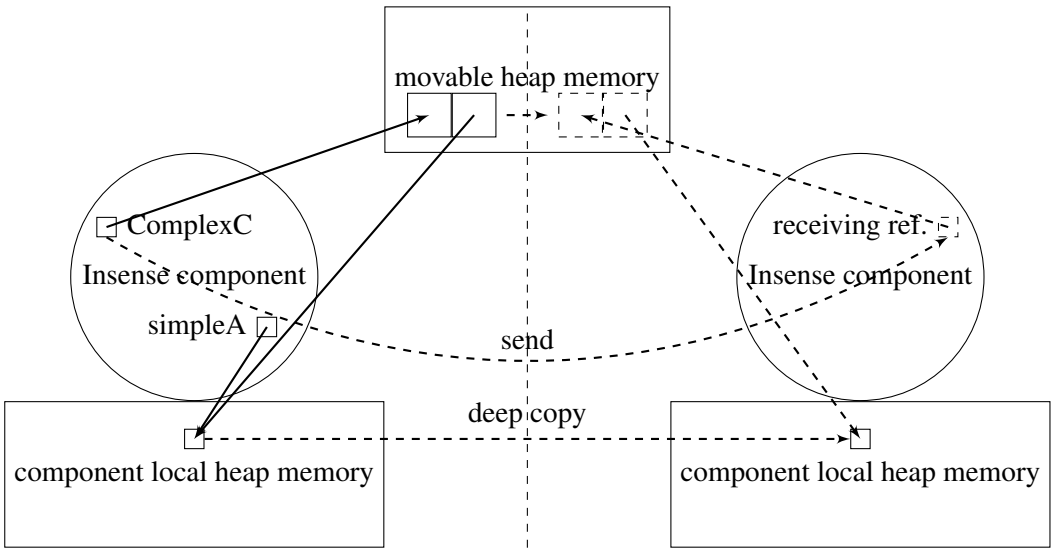


Figure 3.3: $complexC$ case: movable structure with non-movable structure field.

3.1.1.2 Array Semantics

An array in Insense can contain any of the Insense primitive types or a reference to an array or structure. In the following discussion we assume an array element is a reference to a heap-allocated object. The semantics for arrays of primitive types are unchanged.

A flexible array is one in which the size is unknown at compile-time i.e. the array size can be initialised by an expression whose value isn't known until run-time. The presence of flexible arrays in Insense create a problem for the movability analysis in determining how many references exist to a particular object. For example, an array could have all its elements refer to the same object. Further, array indices could be arbitrary expressions meaning we cannot determine at compile-time which array element is being accessed, making it impossible to determine whether there are any accesses to invalid (moved) memory in general.

To handle the complexity, restrictions are placed on arrays, and extensions to the existing Insense array creation mechanisms are provided to enable usage of move semantics. When declaring an array, a template object is provided to the **new** operator. A copy of the template object is made for each element in the array. For non-movable arrays, reference-counting of the array elements is performed and the array is deep-copied on **send** operations. For movable arrays, the array is reference-counted, and individual elements have the same count as the entire array. An assignment in which an array element appears on the right-hand side will increment the count of *the array*. In other words, the left-hand side will alias (see Section 3.3) the array, not the single element. Additionally, you cannot send individual array elements across a channel, only the entire array. The justification for this restriction is that sending the entire array is the most common use case in Insense. Sending a reference that aliases an array will invalidate the entire array.

An array can be declared to contain movable or non-movable objects but not both. Hence, only arrays created in the movable heap (using **mov**) are considered to contain references to movable memory, and all elements must be movable. It is prohibited to assign a non-movable object into an array element of a movable array, or vice-versa. Consider Listing 3.4, where assignments to array elements of objects in different memory spaces results in a compile-time error. Note that the assignment, $c := a[n]$, is valid since this will just decrement the reference count on the movable object referenced by c before the assignment, and increment the reference count on the non-movable object referenced by $a[n]$. Similar for $d := b[n]$.

What if a programmer wants to perform $a[n] := c$? To perform the assignment, the memory referenced

Listing 3.4: Cannot assign movable memory to non-movable array or vice versa.

```
type simpleStruct is struct(integer i);

a = new simpleStruct[n] of simpleStruct(0);
c = mov new simpleStruct(2);

b = mov new simpleStruct[n] of simpleStruct(0);
d = new simpleStruct(2);

a[n] := c; // error, a is an array of references to non-movable memory

b[n] := d; // error, b is an array of references to movable memory
```

Listing 3.5: An example of an interface specifying a channel receiving movable memory.

```
type IRecvComp is interface(in mov simpleStruct refin; // <--- movable heap objects
                            out simpleStruct refout;
                            out integer intout)
```

by c must be copied into non-movable (component local) memory. Annotating the assignment with **copy** ($a[n]$:= **copy** c) instructs the compiler to create a duplicate of the object referenced by c, and set $a[n]$ to reference it. The type of memory returned by the copy operator is inferred by the type of the left-hand side of the assignment. In other words, **copy** preserves the memory space type of the left-hand side. The same rule applies for assignment of non-movable objects to movable array elements.

3.1.2 Channel Semantics

The defintion for channels within an interface is extended to provide the possibility of declaring the memory object being received as movable. Listing 3.5 demonstrates the use of the **mov** annotation in this context. An implementing component will use the **receive** primitive in the regular manner, the only difference being that the received heap object is contained within the movable heap memory space and the analysis will track the reference to ensure the object is only ever moved at most once. Note that it does not matter whether or not a movable was sent from the sender, either a deep-copy was performed or the sender sent the heap object as a movable. In either case, the memory is not owned by any other component and can safely be used by the receiver.

3.1.3 Effect on Behaviour Clause

Any movable memory references that are sent within the behaviour clause, either directly, or indirectly via a procedure call, must be initialised somewhere inside the behaviour clause without any encapsulating control flow construct. The use of the **stop** statement within the behaviour clause does not negate this requirement. For example, Listing 3.6 shows an ill-formed behaviour clause. Even though a will never be sent twice because of the presence of the **stop** statement, the program is not valid since, in general, the presence of a **stop** statement does not guarantee behaviour termination after one iteration (see Listing 3.7).

Listing 3.6: Ill-formed behaviour clause

```
a = mov new Foo;
behaviour {
    send a on chan;
    stop;
}
```

Listing 3.7: Why use of stop does not save programmer

```
a = mov new Foo;
behaviour {
    send a on chan;
    receive v from intchan;
    if v == 1 then {
        stop;
    }
}
```

3.2 Formulating the Move Analysis

This section presents the data flow equations required to track movability of heap objects within an Insense program. In particular, we first define the intraprocedural data flow equations for handling movability, abstracting away the handling of aliasing which is subsequently presented as another set of data flow equations in the following section, then we extend the equations to the interprocedural setting using the theory presented earlier in Chapter 2.

3.2.1 Intraprocedural Move Analysis

The equations act on a data flow framework defined as $(L, F, \cup)$, where L is the bounded meet semilattice of data flow values, F is the set of flow functions operating on elements of L, and $\cup$ is the meet operator to handle control-flow merge points. Our bounded meet semilattice of data flow values is the set of variables that cannot be moved at a specific program point, therefore our greatest element is $\emptyset$ and our least element is the number of reference variables to movable memory in the program, M. We also have that, $M \leq V$, where V denotes the set of all variables in the program. Recall, that a *use* is defined to be any reference to a variable that does not re-define that variable. If an instruction n is $x = y$, then we can say that n is $use(y)$ ("n contains a use of y").

We shall utilise Equation 2.1. We need to define the flow function f_n. The flow functions for move analysis must take into consideration the aliases of references, including global, component member and local variables, and reference formal parameters. For the moment, assume the aliases of all references, and reference formal parameters have been computed for all program points u such that for all references (including reference formal parameters) x, $ALIAS(x, u)$ is the set of references that may be aliases of x upon entry to program point u (see Section 3.3); that is, x may reference the same memory location as the variables in $ALIAS(x, u)$. Then our equation can be formulated as:

$$f_n(x) = (x - Kill_n) \cup Gen_n \qquad \text{if } n \text{ is } use(v), \text{ and } v \in x, \text{ then generate an error} \tag{3.1}$$

where

$$Gen_n = \begin{cases} \{v\} \cup ALIAS(v,n) & \text{if } n \text{ is a send operation sending } v,\ v \in M \\ \{y\} \cup ALIAS(y,n) & \text{if } n \text{ is a send operation sending } y,\ y \text{ is a reference formal parameter} \\ \emptyset & \text{otherwise} \end{cases} \tag{3.2}$$

$$Kill_n = \begin{cases} \{v\} & \text{if } n \text{ is an assignment } v = r,\ v \in M,\ r \in V - (v \cup ALIAS(v,n)) \\ \{y\} & \text{if } n \text{ is an assignment } y = r,\ y \text{ is a reference formal parameter, } r \in V - (y \cup ALIAS(y,n)) \\ \{v\} & \text{if } n \text{ is a receive operation receiving } v,\ v \in M, \\ \emptyset & \text{otherwise} \end{cases} \tag{3.3}$$

The side-effect of $f_n(x)$ is our error checking on the data flow values reaching program point n. Thus, for $x \in L$ and $v \in x$, if at program point n we send v across a channel, an error should be generated since v has already been moved in some path from the entry node of the CFG under consideration to n.

For each program point n, we associate an In and an Out set denoting the data flow values at the entry and exit of program point n, respectively:

$$In_n = \begin{cases} \emptyset & \text{if } n \text{ is the entry node of the CFG} \\ \bigcup_{p \in pred(n)} Out_p & \text{otherwise} \end{cases}$$
$$Out_n = f_n(In_n)$$

Lemma 3.2.1. *Using Algorithm 2, the data flow framework, $(L, F, \cup)$, for intraprocedural move analysis will terminate after at most $d(G) + 3$ iterations for every instance of the problem.*

Proof. Since $f : L \mapsto L$ can be written in the form $f(x) = (x - Kill) \cup Gen$, that is, as a bit-vector framework [KSK09] it follows that $(L, F, \cup)$ is distributive [KU76]. From Lemma 2.1.5 and Theorem 2.1.4 we have that our data flow framework for intraprocedural move analysis satisifies the rapid condition and thus will terminate after at most $d(G) + 3$ iterations for every instance of the problem. □

3.2.2 Interprocedural Move Analysis

To extend the move analysis to the interprocedural setting we utilise the call-strings approach described in Section 2.1.2.2 and extend our data flow equations to qualify data flow values based on the calling contexts. The definitions in Section 2.1.2.2 hold for the move analysis and we simply need to plug our flow function f_n into Equation 2.3. Call and return nodes affect propagation of movability through aliasing of actual and formal parameters, we discuss this issue in Section 4.4.2.3.

3.3 Alias Analysis

Consider a control flow graph, $G = (N, E)$, with entry node n_{entry}. At some node $n \in N$, two program reference variables, x and y, are *may-aliased* to each other at n if they refer to the same object in *at least one* path from n_{entry} to n. If x and y refer to the same object in all paths from n_{entry} to n, then we say x and y are *must-aliased*. In this work we focus our attention on may-alias information.

In order to track all references that could potentially refer to movable memory we must define a data flow framework for computing aliases. The framework for alias analysis is loosely based on the flow-sensitive *may*-alias analysis work described by Choi, Burke, and Carini [CBC93]. Where our approach differs is in the use of the control flow graph rather than the sparse evaluation graph (SEG), and performing interprocedural alias computation using the call-strings approach as opposed to realisable execution paths and alias instances. Our approach has the advantage that the extension to the interprocedural case is simpler than propagating alias instances, requiring minimal changes to the definitions of Gen and $Kill$. While using the SEG would be less computationally expensive than using the CFG, the size of Insense procedures and hence the number of nodes in a CFG, in general, is likely to be very small. We shall now discuss the intraprocedural and interprocedural cases separately.

3.3.1 Intraprocedural Alias Analysis

We define an alias pair, $< x, y >$ for program reference variables x and y, to denote the possibility that x and y may refer to the same memory location at some program point. The data flow equations for alias analysis act on a data flow framework defined as $(L_{AA}, F_{AA}, \cup)$, where L_{AA} is the meet semilattice of data flow values, F_{AA} is the set of flow functions operating on elements of L_{AA}, and $\cup$ is the meet operator to handle control-flow merge points. The elements of the meet semilattice are sets of alias pairs at a specific program point, therefore our greatest element is $\emptyset$ and our least element is the cartesian product of the number of variables in the program, $V \times V$.

Following [CBC93], we can define the analysis using In and Out sets for each program point, noting that only assignment statements to references and call sites modify the sets. We shall defer discussion of call sites for the moment as this is handled by the interprocedural phase of the analysis.

Let $g : L_{AA} \mapsto L_{AA}$ be the flow function for tracking aliases. Then

$$g_n(x) = (x - Kill_n(x)) \cup Gen_n(x) \tag{3.4}$$

where

$$Gen_n(x) = \begin{cases} \{(v, u)\} & \text{if } n \text{ is } v := u,\ u, v \in V \\ \emptyset & \text{otherwise} \end{cases} \tag{3.5}$$

$$Kill_n(x) = \begin{cases} \bigcup\limits_{w \in V} \{(v, w), (w, v)\} & \text{if } n \text{ is } v := u \text{ and } (u, v) \notin x, \text{ where } u, v \in V \\ \emptyset & \text{otherwise} \end{cases} \tag{3.6}$$

Equation 3.5 says that upon reaching an assignment or declaration statement, $n : v := u$, the alias pair (v, u) is added to the gen set for program point n. Note, the aliases of v before the assignment do not become aliased to u since $v := u$ is equivalent to a pointer assignment in C. Equation 3.6 says that upon reaching a statement, $n : v := u$, the analysis adds all alias pairs involving v to the kill set for program point n. If v is involved in any other alias pair as a prefix, it is replaced by one of its former aliases in the alias pair.

In Section 3.2.1 we defined the set $ALIAS$ for each pair, program point n and variable $v \in V$; it is the set of the *access paths* that are *may*-aliased to each heap object at the particular program point. For Insense, an access path is an expression combining variable names, subscript operators, and field selectors such that the expression evaluates to a reference. Equation 3.4 does not provide the set $ALIAS$ but the set of alias pairs valid at a program point. The move analysis requires the alias mapping so we must define $ALIAS$ formally from the set of alias pairs provided by our meet semilattice L_{AA}. Now, we make our definition of $ALIAS$ more concrete:

Defintion 3.3.1. The set $ALIAS(v, n)$ returns the *transitive closure* of the alias pairs of v upon entry to program point n. Formally, $u \in ALIAS(v, n)$ if and only if there exists alias pairs $(v, y_1), (y_1, y_2), \ldots, (y_n, u) \in x$ at program point n, where $x \in L_{AA}$ and represents the set of data flow values entering program point n, and $v, y_1, y_2, \ldots, y_n, u \in V$.

3.3.2 Interprocedural Alias Analysis

Using the call-strings approach, we extend the alias analysis to handle call and return nodes created for each call site; these nodes generate and kill aliases, between actual and formal reference parameters, respectively. We amend Equations 3.5 and 3.6 to handle these interprocedural cases. $IntraGen_n$ and $IntraKill_n$ handle the intraprocedural cases described in Section 3.3.1. Given actual and formal parameters, $a_i, f_i \in V$ for $1 \leq i \leq n$ ($n \in \mathbb{N}$), and $u, v \in V$, we have:

$$Gen_n(x) = \begin{cases} \bigcup_{i=1}^{n} \{(a_i, f_i)\} & \text{if } n \text{ is a call node for procedure call } p(a_1, a_2, \ldots, a_n), \\ \bigcup_{i=1}^{n} \bigcup_{(f_i,u),(v,f_i) \in x \wedge v \neq u} \{(u, v)\} & \text{if } n \text{ is a return node for procedure call } p(a_1, a_2, \ldots, a_n), \\ \{(v, p(a_1, a_2, \ldots, a_n))\} & \text{if } n \text{ is return statement returning } v \text{ for procedure call } p(a_1, a_2, \ldots, a_n), \\ IntraGen_n(x) & \text{otherwise} \end{cases} \tag{3.7}$$

$$Kill_n(x) = \begin{cases} \bigcup_{i=1}^{n} \bigcup_{w \in V} \{(f_i, w), (w, f_i)\} & \text{if } n \text{ is a return node for procedure call } p(a_1, a_2, \ldots, a_n), \\ IntraKill_n(x) & \text{otherwise} \end{cases} \tag{3.8}$$

Equation 3.7 generates alias pairs between actual and formal parameters upon entry to the call node for a procedure call site, propagates any aliases created, to formal parameters, during the analysis of a call upon entry to the return node for a procedure call site, and handles aliasing of returned references with their respective call sites (see Section 4.4.1.2 for a more concrete description).

Equation 3.8 kills all alias pairs involving formal parameters upon entry to the return node for a procedure call site. For cases where there are no parameters to the procedure call, no alias generation or killing is performed.

Chapter 4

Implementation

This chapter discusses the implementation details of the compiler and, in particular, the move and alias analyses.

4.1 Implementation choices

The compiler frontend that formed the basis for this work was implemented in C, therefore C was chosen for the implementation language for the compiler middle-end modules. The entire compiler middle-end was written in approximately 7,000 lines of code and the data flow analyses (framework, alias analysis, and move analysis) constitute 1,493 lines of the total (including unit tests) [1].

4.2 Intermediate Representation

The result from the frontend of the Insense compiler is an abstract syntax tree of the program. This form is not suitable for program analysis since it does not represent control flow explicitly. Further, expressions may be of arbitrary length, and the structure of expressions defined by the grammar presents too much irrelevant detail. This section describes the intermediate representation on which the analyses operate. In the first section we describe the main constructs used to represent entire Insense programs, components, functions, and interfaces. In the following section we describe the instructions and values within the intermediate representation.

4.2.1 Representing Programs, Components, and Functions

The IR defines a representation for the entire program, enabling the later stages of the middle-end to obtain information about a variety of program properties by querying well-defined interfaces. The **IRProgram** ADT describes an entire Insense program; it holds mappings for all components, globally defined procedures, and interfaces defined within the program. The mappings map names to IR ADT instances (**IRComponent**, **IRFunction**, or **IRInterface**, which are described below). Additionally, it maintains the program call graph (see Section 4.4.2.1) and the control flow graph for the primordial main (component initialisation, connection statements, etc).

[1] Calculated using Wheeler's SLOCCount tool [Whe14].

Kind	Description	Form	Uses
assign	Assignment	*a := b*	uses(a), b, uses(b)
call	Procedure call	*f(*a_1*,* a_2*, ...)*	a_1, uses(a_1), a_2, uses(a_2), ...
cjump	Conditional jump	*cjump cond,B1,B2*	$cond$, uses($cond$)
connect	Connect statement	*connect* a *to* b	a, uses(a), b, uses(b)
decl	Declaration statement	a = b	b, uses(b)
disconnect	Disconnect statement	*disconnect* a	a, uses(a)
jump	Unconditional jump	*jump B1*	-
nop	No operation	*nop*	-
recv	Receive statement	*receive* a *from* b	b, uses(b)
ret	Return statement	*return* r	r, uses(r)
send	Send statement	*send* a *on* b	a, uses(a), b, uses(b)
stop	Stop statement	*stop*	-

Table 4.1: List of available IR instructions and their internal uses.

The **IRComponent** ADT defines an Insense component which stores the local declarations of program variables (in a separate CFG) and the local procedure definitions in a mapping from their name to an **IRFunction** instance.

The **IRInterface** ADT describes an Insense interface. The typed channels defined by the interface are maintained in a mapping; components implementing the interface may query this mapping to obtain the channel's type information e.g. whether a certain **in** channel receives movable memory or not.

The **IRFunction** ADT defines an Insense procedure; it maintains a list of the formal parameters to the procedure and the control flow graph representing the body of the procedure.

4.2.2 Representing Instructions

To simplify the representation for the analysis, an intermediate representation was defined closely modelling Insense constructs, but deconstructing control flow constructs into simple conditional and unconditional jump instructions. Expressions are decomposed so that at most three operands appear in one instruction along with a binary operator. Note, however, the IR is not technically three-address code since some constructs, such as procedure calls, are not deconstructed. Additionally, some expressions, such as array and structure dereference expressions, are not deconstructed to provide the analysis a simple method for tracking composite movables; simply having access to the left and right hand side of an expression to compare movable status. This approach is taken since we must restrict array assignments to comparable memory types (see Section 3.1.1.2), so the analysis has to take those cases into account which cannot be adequately determined by the frontend type checker such as the movable status of a function call, for example.

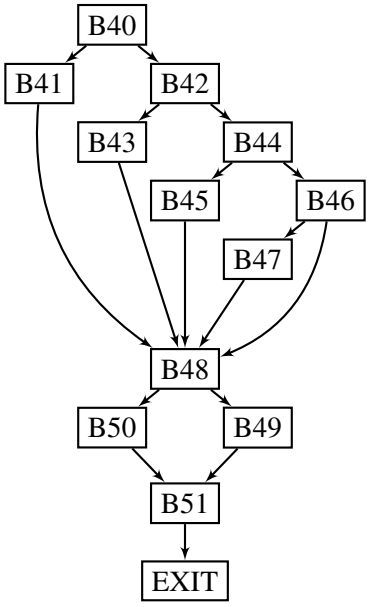
The IR instructions are collected together into basic blocks. The exit point at the end of the block is either a conditional or unconditional jump to at most two blocks. Each block may have any number of predecessor blocks. An entire Insense procedure is represented by a CFG. Table 4.1 gives an overview of the instructions in the intermediate representation. Figure 4.1(a) shows an Insense code snippet, Figure 4.1(b) shows the corresponding Insense IR upon transformation (basic block labels have been added for clarity), and Figure 4.1(c) displays the corresponding control flow graph segment.

```
if val == 0 then {
        printString("Rec:␣got␣tic\n");
} else if val == 1 then {
        printString("Rec:␣got␣toc\n");
} else if val == 2 then {
        printString("Rec:␣got␣tac\n");
} else if val == 3 then {
        printString("Rec:␣got␣def\n");
}
if val == 0 then {
        printString("Rec:␣got␣tic\n");
} else {
        printString("Rec:␣got␣toc\n");
}
```

(a) Insense Source

```
B40:
t20 := val == 0
if t20 then goto B41 else goto B42
B41:
printString("Rec:␣got␣tic\n")
goto B48
B42:
t21 := val == 1
if t21 then goto B43 else goto B44
B43:
printString("Rec:␣got␣toc\n")
goto B48
B44:
t22 := val == 2
if t22 then goto B45 else goto B46
B45:
printString("Rec:␣got␣tac\n")
goto B48
B46:
t23 := val == 3
if t23 then goto B47 else goto B48
B47:
printString("Rec:␣got␣def\n")
goto B48
B48:
t24 := val == 0
if t24 then goto B49 else goto B50
B49:
printString("Rec:␣got␣tic\n")
goto B51
B50:
printString("Rec:␣got␣toc\n")
goto B51
B51:
goto EXIT
```

(b) Insense IR (textual)



(c) Insense IR (graphical)

Figure 4.1: Transformation from Insense to Insense IR

Description	Form	Uses
Array construction	*new* <*type*> $[n_1][n_2]\cdots[n_k]$ *of* $init$	uses(n_1), uses(n_2), ..., uses(n_k), uses($init$)
Binary operation	a *op* b	a, b, uses(a), uses(b)
Channel construction	*new out integer*	-
Structure construction	*new* <*type*> $(v_1, v_2, \ldots)$	uses(v_1), uses(v_2), ...
Copy Operation	copy v	v, uses(v)
Array subscript expression	$v_1[v_2]$	abs_base(v_1), uses(v_1), v_2, uses(v_2)
Structure field expression	a.x	abs_base(a), uses(a)
Constant integer	4	-
Identifier	v	-
Literal	v	-
Channel selection (select statement)	*select* $\{ticker, tocker, tacker\}$	$ticker$, $ticker$, $tacker$
IR Temporary	t4	-
Unary operation	*unary_op* v	v, uses(v)
Function call	f(*a_1, a_2, ...*)	a_1, uses(a_1), a_2, uses(a_2), ...

Table 4.2: List of available values and their internal uses.

4.3 Value Representation

We need to represent program variables, and other values, within the IR instructions. Since the data flow analyses require a representation for program variables, it was convenient to use the same representation within IR instructions as for the data flow values rather than the more commonly used bit-vector representation. The additional space overhead within the data flow sets is not onerous, and provides efficient access to various properties of program variables that can be used in the analysis to provide detailed error generation and obtain memory type information.

Table 4.2 summarises the possible values that can appear in an Insense program. Notice that a call is represented both as an IR instruction and a value. The value encodes Insense procedures that return values whereas an IR call instruction corresponds to a void procedure. To distinguish between the two, we use the term *function* to describe a routine returning a result and *procedure* to describe a routine returning void.

4.4 Data Flow Analysis

The computation of data flow information for movability and aliases required specification of the intraprocedural data flow equations and transliteration of the algorithms presented in Section 2.1.2.

4.4.1 Implementing Intraprocedural Analysis

The data flow analysis is implemented in two parts. First, an alias analysis pass computes for all program variables their set of may-aliases at each program point. Second, the move analysis pass performs movability tracking and error detection on the program to determine whether or not it is sound. Both of these analyses have interprocedural and intraprocedural components.

A common framework for the iterative intraprocedural data flow analysis algorithm (Algorithm 2) was developed. Algorithm 3 illustrates the general data flow analysis framework for computing information for procedures. The function, $has\_changed$, is a predicate returning $true$ if the data flow values have changed during the previous iteration, $false$ otherwise. An analysis will provide a function to implement the $has\_changed$ predicate. The advantage of leaving the details of the predicate to each analysis allows analysis-specific stopping conditions (see Section 4.4.1.1). The functions, $basic\_block\_init$ and $basic\_block\_mfp$, compute the initial and maximum fixed point for the data flow values, respectively.

Algorithm 3: General data flow algorithm for computing MFP assignment on a CFG.

```
1 void dfa(CFG G)
2 begin
3     B ← reverse postorder of blocks in G
4     for b ∈ B do
5         basic_block_init(b)
6     while has_changed() do
7         for b ∈ B do
8             basic_block_mfp(b)
```

Although the implementation of these functions are specific to an analysis, the general theme for $basic\_block\_init$ and $basic\_block\_mfp$ is that depicted in Algorithm 4 and Algorithm 5, respectively. The main idea is for Algorithm 4 to compute the data flow values on entry to a basic block from only those basic blocks which precede the block in the reverse post-ordering. In other words, the initial phase excludes back edges since the blocks at the source of these edges will not have valid data flow information yet. Algorithm 5 will alter the return value of the predicate if any data flow sets change during the iterative computation. In these algorithms we have assumed, for simplicity of the exposition, that the predicate returns some globally accessible boolean, $changed$, initially set to $true$.

Algorithm 4: General approach for computing initial data flow values of a basic block.

```
1 void basic_block_init(BasicBlock b)
2 begin
3     P ← predecessor blocks of b
4     for p ∈ P do
5         if rpo_number(p) < rpo_number(b) then
6             In_b = In_b ⊓ f_p(In_p)
```

Algorithm 5: General approach for computing MFP data flow values of a basic block.

```
1 void basic_block_mfp(BasicBlock b)
2 begin
3     P ← predecessor blocks of b
4     temp ← ∅
5     for p ∈ P do
6         temp = temp ⊓ f_p(In_p)
7     if temp ≠ In_b then
8         In_b = temp
9         changed = true
```

The common framework accepts a state instance to all the interface functions enabling each analysis to define

its own state structure for use within its implementations of the interface functions. We shall discuss the specifics of the intraprocedural move and alias analyses in the following two sections.

4.4.1.1 Move Analysis

The state structure provided for the move analysis pass is the **MAS** type which maintains information such as special flags, a **DataFlowSet** instance, a **Context** instance, an **IRPRogram** instance, an alias mapping, and a **MovableSet** instance.

The **MovableSet** instance maintains the set of values which correspond to references pointing to movable memory. It is updated after each instruction during the analysis, and initialised upon entry to every local procedure of a component with the component's local declarations. Later, an algorithm will be presented to show how the set is affected by an IR instruction.

Prior to starting any analysis, the module computes the aliases for the program using the alias analysis pass (see Section 4.4.1.2) and stores the result in the alias mapping field of a **MAS** instance.

The **IRProgram** instance is stored within the state structure to provide access to whole-program information such as the program's call graph.

The **Context** instance stores the current calling context of the analysis.

The **DataFlowSet** instance maintained by the **MAS** type is used to store the data flow values being computed for the current CFG of interest.

The special flags maintained by the **MAS** type are to control the $has\_changed$ predicate. The changed flag indicates whether the data flow values, associated with a CFG's basic blocks, have been modified during the last iteration. The error flag indicates whether a move error was detected during the analysis of a CFG. The move analysis exits the iterative computation on a CFG early when a move error is detected, preventing spurious error messages being generated due to the propagation of invalid data flow information.

Algorithm 6 depicts the algorithm for the move analysis flow function which computes the moved variables at the end of the specified block. The algorithm traverses the instructions of the block b computing the data flow values present at the end of the block (Out_b) and updates the current set of references to movable memory (see below and Algorithm 7). The functions $compute\_gen$ and $compute\_kill$ implement Equations 3.2 and 3.3 respectively. Discussion of the interprocedural effects computed by Algorithm 6 is deferred until Section 4.4.2.3. Algorithm 6 handles checking for array assignment cases that cannot be detected by the frontend type checker. In particular, if an array element is assigned the result of a function call we cannot know prior to performing data flow analysis whether the returned reference points to movable memory or not. Checking for invalid array assignments involves analysing the instruction using aliasing information known about its operands. An error message, which takes a similar form to the one produced by the frontend for the trivial cases, is displayed if any invalid assignments are detected and the error flag is set in the **MAS** instance.

Algorithm 7 provides the IR instruction cases which require modification of the movable set. The cases for assignment and declaration are straightforward. An **in** channel annotated with the **mov** keyword will receive movable memory, so the declared variable within the **receive** primitive must be added to the movable set. Note for return IR instructions, the value representing the call from the call site is added to the movable set if the called function returns a movable memory object, so as to propagate the movability information back to the caller.

Algorithm 6: Algorithm for the move analysis flow function.

```
1  void compute_moved_values(MAS mas, BasicBlock b)
2  begin
3      for inst ∈ b do
4          if inst does not contain a move error then
5              if inst contains procedure call then
6                  compute effect of procedure call on data flow values
7                  if move error detected then return
8              if inst contains invalid array assignment then return
9              Gen ← compute_gen(mas, inst)
10             Kill ← compute_kill(In_b, inst)
11             Out_b ← (In_b − Kill) ∪ Gen
12             update_movables_set(mas, inst)
           else
13             return
```

Algorithm 7: Algorithm for updating the movable set.

```
1  void update_movables_set(MAS mas, IRInst inst)
2  begin
3      MS ← get_movable_set(mas)
4      switch inst do
5          case a := b
6              if a is not movable and b is movable then
7                  MS ← MS ∪ {a}
8              if a is movable and b is not movable then
9                  MS ← MS \ {a}
10         case receive v from chan
11             if chan receives movable memory then
12                 MS ← MS ∪ {v}
13         case a = b
14             if a is not movable and b is movable then
15                 MS ← MS ∪ {a}
16         case return r
17             if r is movable then
18                 Ctx ← get_calling_context(mas)
19                 if Ctx has at least one edge then
20                     site ← retrieve call site of first edge in Ctx
21                     vc ← value representing the function call of site
22                     MS ← MS ∪ {vc}
23         otherwise
               return
```

4.4.1.2 Alias Analysis

The state structure provided for the alias analysis pass is the **AAS** type which maintains information such as special flags, a **DataFlowSet** instance, a **Context** instance, an **IRPRogram** instance, and an alias mapping. The usage of these instances is similar to the use cases for the **MAS** type. Of note however, is the alias mapping field of an **AAS** instance which stores the mapping to be returned from the analysis. The $has\_changed$ predicate for alias analysis is based solely on whether the data flow values, associated with a CFG's basic blocks, have been modified during the last iteration.

The alias analysis pass computes for each instruction (not basic block) the may-alias pairs present at that program point. We must store sets of alias pairs per instruction rather than per basic block since aliases are altered at the instruction level and the alias information for each instruction is needed for movability error checking. Thus, the In and Out sets defined in Equation 3.4 refer to the alias pairs which hold on entry to and on exit from an IR instruction.

The alias analysis flow function simply computes the effect of all instructions in the provided basic block. Algorithm 8 depicts the algorithm for computing the effect of an instruction on the input data flow set. The $retrieve\_aliasable\_value$ function handles dereference expressions such as array subscript expressions, whose aliasable value is the entire array (see Section 3.1.1.2), and structure field expressions which do not have aliasable values (see Chapter 6). For regular program reference variables, $retrieve\_aliasable\_value$ returns the value corresponding to the reference variable itself.

Algorithm 8: Algorithm for the computing effect of an instruction on alias analysis data flow sets.

```
1  void compute_inst_effect(AAS aas, IRInst inst, DataFlowSet DFS)
2  begin
3      switch inst do
4          case a := b or a = b
5              lhs ← retrieve_aliasable_value(a)
6              rhs ← retrieve_aliasable_value(b)
7              if lhs and rhs are valid values then
8                  for every alias pair (x, y) ∈ AS such that x = lhs or y = lhs do
9                      DFS ← DFS \ {(x, y)}
10                 DFS ← DFS ∪ {(lhs, rhs)}
11         case return r
12             Ctx ← get_calling_context(aas)
13             v ← retrieve_aliasable_value(r)
14             if v is a valid value and Ctx has at least one edge then
15                 site ← retrieve call site of first edge in Ctx
16                 if site is an assignment or declaration IR instruction then
17                     vc ← value call of site
18                     DFS ← DFS ∪ {(r, vc)}
19         otherwise
               return
```

4.4.2 Implementing Interprocedural Analysis

As described in Section 2.1.2.2, interprocedural data flow analysis considers the effect of procedure call instructions within an Insense program. The call-strings approach [SP81] maintains a stack of uncompleted procedure calls. Upon the discovery of a call, the callee must be analysed with the incoming data flow information provided by the caller, and the synthesised information passed back to the correct call site. In other words, data flow information should be propagated along interprocedurally valid paths only. While the literature defines the call-strings approach in terms of analysing the program supergraph [Mye81], the implementation in the Insense compiler extends the intraprocedural data flow analyses to handle IR procedure call instructions and function call values by utilising the properties of the program call graph to connect callees to their callers. In the following sections, we expand on the main implementation issues arising from adding interprocedural analysis to the Insense compiler.

4.4.2.1 Call Graph

To facilitate the interprocedural analysis in the Insense compiler, the program's call graph is constructed. For Insense, the call graph consists of nodes representing global procedures, component procedures and behaviour clauses of every component, and the built-in functions defined by the standard library. Each call in the program is represented in the graph as an edge, $p \xrightarrow{c} q$, where caller p calls q at call site c. The graph maintains information of each call so that the actual parameters can be substituted into the function, in place of the formal parameters, during the analysis. The call graph construction algorithms for an entire Insense program, component, and function are shown in Algorithms 9-11, respectively. A node in the call graph is created for **IRFunction**s during their first involvement in edge creation. From the algorithms, and remembering that Insense does not have recursive procedures, we can see that construction of the program call graph is bounded by $O(|C||LP_C| + |GP|)$ where $|C|$ denotes the number of components in the program, $|LP_C|$ denotes the largest number of procedures defined locally within a component $c \in P$, and $|GP|$ denotes the number of globally defined procedures.

Algorithm 9: Constructing the call graph from an Insense program's IR.

```
1 void program_build_call_graph(CallGraph G, IRProgram P)
2 begin
3     for every Insense component C ∈ P do
4         component_build_call_graph(G, C)
5     for every globally defined function F ∈ P do
6         function_build_call_graph(G, F)
```

Algorithm 10: Constructing the section of the program call graph of an Insense IR component.

```
1 void component_build_call_graph(CallGraph G, IRComponent C)
2 begin
3     for every locally defined procedure P ∈ C do
4         function_build_call_graph(G, P)
```

In addition to an edge in the call graph being associated with the caller, the callee, and the call site, each edge has a *payload* to which an analysis can attach information that it wishes to propagate between caller and callee. The payload for both the alias and move analyses are the set of data flow values on entry and exit to the call.

Algorithm 11: Constructing the section of the program call graph of an Insense IR function.

```
1 void function_build_call_graph(CallGraph G, IRFunction F)
2 begin
3     CFG ← get_function_cfg(F)
4     for every instruction I ∈ CFG do
5         if I is an IR call instruction calling function Q then
6             create an edge F --I--> Q in G
7         else if I contains a value representing a call to function Q then
8             create an edge F --I--> Q in G
```

4.4.2.2 Calling Contexts

To handle interprocedural analysis, a calling context is maintained by each analysis. The context is a stack of uncompleted procedure calls. Upon the discovery of a call, the corresponding edge in the call graph is pushed onto the stack and the function is analysed. Thus, the data flow sets contain qualified data flow values and a meet over the values in the sets must have matching contexts before they are combined (using the $\uplus$ operator defined in Section 2.1.2.2).

4.4.2.3 Move Analysis

Algorithm 12 shows the algorithm for generating a fully context-sensitive error message from the move error created from using v in the $inst$ instruction. We can obtain line number information from the IR instruction, and use the edges of the call graph to provide context in the form of procedure calls leading to the error; the edges in the calling context are traversed in FIFO order. Line number information for each call site can be obtained from the stored instruction in each call graph edge. Providing such detailed error messages to the (likely, non-specialist) programmer will aid in resolving issues with movability.

Algorithm 12: Move error message generation algorithm.

```
1 void generate_error_message(MAS mas, IRInst inst, Value v)
2 begin
3     Ctx ← get_calling_context(mas)
4     output ← "error: invalid use on line " + inst_linenum(inst) + v + " has been moved."
5     for edge ∈ Ctx do
6         callee ← cgedge_callee(edge)
7         caller ← cgedge_caller(edge)
8         site ← cgedge_call_site(edge);
9         output ← " in call to " + callee + " from " + caller + " on line " + inst_linenum(site)
```

Algorithm 13 shows the propagation of movability status upon encountering a call site. A formal parameter of a callee is added to the current set of references to movable memory if and only if their corresponding actual parameter at the call site is in the movable set. Algorithm 14 shows the propagation of movability status upon returning from analysing a procedure call. We must remove all formal parameters of the callee from the movable set and the set of data flow values that will be propagated back to the caller. The latter is required since any

moved formal parameters really correspond to moved actual parameters, and formal parameters must have their state reset after every call.

Algorithm 13: Propagating move analysis data flow information upon entry to a call site.

```
1 void handle_movability_on_call(IRFunction Callee, MovableSet MS)
2 begin
3     for every (actual, formal) parameter pair (a, f) of Callee do
4         if a ∈ MS then
5             MS ← MS ∪ {f}
```

Algorithm 14: Propagating move analysis data flow information upon return from a call.

```
1 void handle_movability_on_return(IRFunction Callee, MovableSet MS, DataFlowSet DFS)
2 begin
3     for every formal parameter f of Callee do
4         MS ← MS \ {f}
5         DFS ← DFS \ {f}
```

4.4.2.4 Alias Analysis

Algorithm 15 shows the creation of aliasing between actual and formal parameters upon encountering a call site. Algorithm 16 shows the handling of the return from analysing a procedure call. We must propagate any aliases created within the function involving the formal parameters since these are aliased to the actual parameters. This propagation step involves obtaining the transitive closure (see Defintion 3.3.1) of aliases of each formal parameter then adding these implicit aliases found through transitivity to the alias set. The $transitive\_closure$ function returns the transitive closure of the specified formal parameter. Trivial aliases such as an alias with oneself are not added. Finally, we remove any aliasing associated with the formal parameters in preparation for any further calls of the callee.

Algorithm 15: Propagating alias analysis data flow information upon entry to a call site.

```
1 void handle_aliasing_on_call(IRFunction Callee, AliasSet AS)
2 begin
3     for every (actual, formal) parameter pair (a, f) of Callee do
4         AS ← AS ∪ {(a, f)}
```

4.5 Correctness and Testing

A unit testing framework was developed to test the individual modules but also to test integration between modules. The integration modules represent phases in the compiler middle-end, for example the AST to IR transformation phase, or the move analysis pass. The framework operates by defining for each unit test a set of functions to run and the output specifies any error messages from test asserts and a summary of how many tests passed or failed.

In order to reason about the correctness of the implementation of the analysis we must do a case analysis on the instructions and values to ensure that the analysis will never accept an invalid program as valid, although it

Algorithm 16: Propagating alias analysis data flow information upon return from a call.

```
1  void handle_aliasing_on_return(IRFunction Callee, AliasSet AS)
2  begin
3      for every formal parameter f of Callee do
4          aliases ← transitive_closure(AS, f)
5          for every v ∈ aliases do
6              for every u ∈ aliases do
7                  if v ≠ u then
8                      AS ← AS ∪ {(v, u)}
9      for every formal parameter f of Callee do
10         for every alias pair (x, y) ∈ AS such that x = f or y = f do
11             AS ← AS \ {(x, y)}
```

Listing 4.1: Movable Insense program rejected by compiler.

```
10 component MovComp presents IMovComp {
11     a = new simpleStruct(0);
12     b = mov new simpleStruct(0);
13
14     behaviour {
15         x = a;
16         if false then {
17             a := b;
18         }
19         send a on chan;
20         x.i := 75; // error: invalid use on line 20, 'x' has been moved.
21         send n on waitchan;
22         stop;
23     }
24 }
```

may reject a valid program. To see this, consider the program snippet, and corresponding compilation error, in Listing 4.1 where a initially references non-movable memory. Clearly, the program will never branch into the then clause of the if statement yet this program is rejected since, in general, the conditioning will not be statically determinable. Special cases such as this example could be allowed by the compiler if the compiler is extended to perform a conditional constant propagation data flow analysis with "type determination" [WZ91].

This strategy provides a safe approximation and is similar to how code optimisation is approximated for regular data flow analysis. The approach taken was to formalise for each Insense IR instruction the set of values that it uses, and then to show that the implementation checks all these cases by writing regression tests for them in the test framework. Indeed, in some instances, this will create stronger assertions than required since the list of uses of a particular value could be empty in the case of values that cannot be further decomposed e.g. identifier values. It follows then that there is no use that could follow a send which would result in an invalid memory access.

For each IR instruction, we define a set of uses for the instruction. For each value kind defined by the IR, we define a set of uses for the value. This information is tabulated in Table 4.1 and Table 4.2, respectively. The function, uses(x), gathers the uses of a value. In the tables, a, b, $cond$, r, and v represent values. $B1$, and $B2$ represent basic block identifiers.

The uses of an instruction are captured by the values. To obtain the uses within a value, $uses(x)$ returns the list of values used within the value x, if x contains sub-values as part of its definition, otherwise it returns an empty list. The recursive nature of the value definition is necessary to facilitate correct analysis on array subscript expressions since they must appear in all assignments in which they are involved in order to be checked for valid move semantics, and, therefore, they must not be decomposed. If they were, it would not be possible to infer any assignments to the array by virtue of the straight-line sequence of code.

In Table 4.2, v_1 in the array subscript case will be either a primitive (temporary, or identifier), or it will be another dereference value. In the former case, we can return v_1 as the single use. In the latter case, we must find the absolute base of the array, abs_base(v_1), and return it as well. To achieve this, we need to return all intermediate uses, e.g. $a[b.x][c.y][d.z]$ shall return $\{a[b.x][c.y], d.z, \{a[b.x], c.y, d, \{a, b.x, c, \{b\}\}\}\}$ where the nesting is used to illustrate the recursive calls to $uses$ to obtain sub-values. Structure field expressions are handled similarly. Constants, identifiers, literals, and temporaries are single values and so have no sub-value uses. A channel selection value encodes the non-determinism in an Insense select statement which will choose between the channels specified in the value.

Chapter 5

Evaluation

This chapter evaluates the space savings on non-trivial Insense applications. We present the methodology and the performance results.

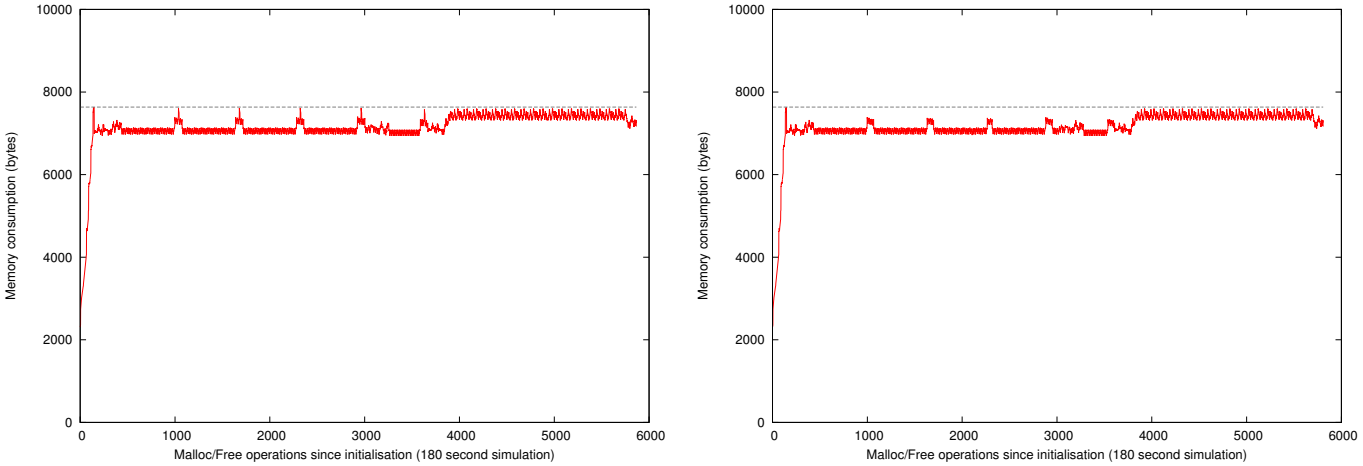
5.1 Methodology

We evaluate our system on a number of Insense applications; some are real applications, others are contrived for the purposes of highlighting the benefits of movability. We compare our empirical results with previous work [CHS13].

5.2 Performance Measurements

The river cats program consists of two T-mote Skys, a base station, and a data logger. The base station sends commands to the data logger obtained from a serial connection with a PC. The data logger samples from its sensors and temporarily stores them on local flash storage to be sent to the base station upon receipt of a "dump" command. The data logger is implemented using three Insense components: a controller component that waits for commands from the base station; a sensor sampler component that periodically samples from the T-mote's sensors storing the results in an array before sending the array to the data gatherer component; and, a data gatherer component that waits to be sent the samples from the sampler and then sends them to the component's flash storage component which is defined by the standard library. The experiment in [CHS13] on the river cats program was repeated, and the memory usage graph for the data logger is shown in Figure 5.1(a). Compare the memory usage of this program with that of the modified river cats program used to take advantage of move semantics (Figure 5.1(b)). From the graphs we see that from allocations 500 to 3500 the spikes in memory usage are less severe in the case of the movable version. To achieve this performance improvement, the data logger's **in** channel from the sensor sampler was marked as receiving movable memory thus there is no copy when the data logger sends the data to local flash storage. Figure 5.1(c) highlights (in yellow) the change to the river cats program necessary to provide the efficiency savings of the movable version over the non-movable version. The design of the application was amenable to the movability property, requiring only a single addition of the **mov** annotation to specify that an **in** channel receives movable memory therefore removing the need for a deep-copy when the data received is subsequently sent on.

A typical example where movability is helpful occurs when there are intermediary component(s) between a sender and a receiver, and the intermediary does not require its own copy of the data being sent. Marking the **in**



(a) Memory usage graph for the original (non-movable version) River cats program.

(b) Memory usage graph for the movable version of the River cats program.

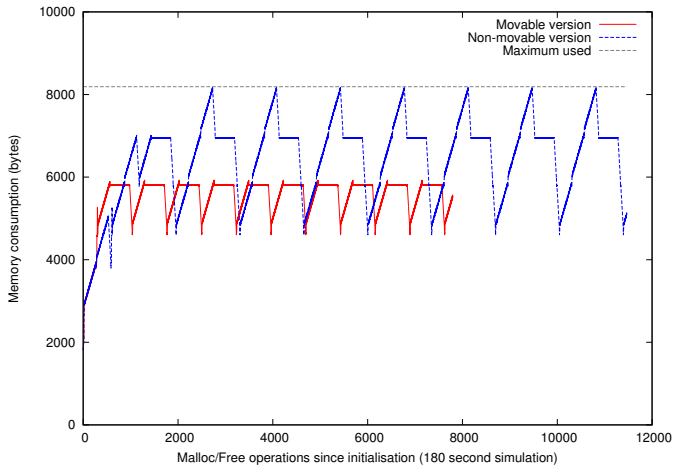
```
52 type IDataGather is interface(in bool controller; out bool sas;
53                               in mov integer[] sasDone; out integer[] writeChan)
```

(c) Source code change required to enable efficiency saving.

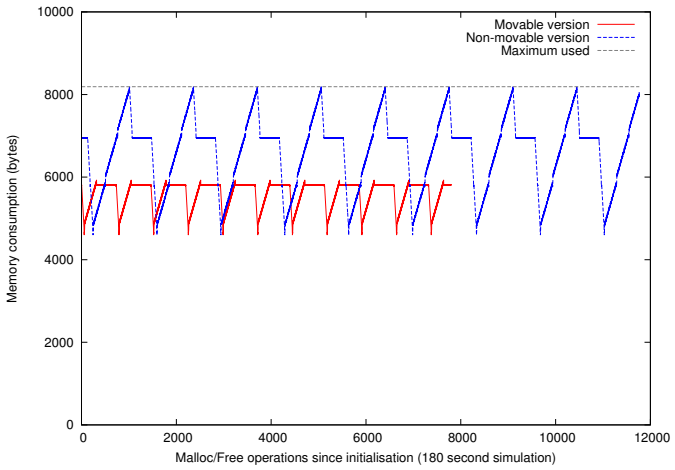
Figure 5.1: Memory usage graphs for river cats program.

channels of all intermediatries as movable prevents the need to create a copy, and the memory reference can be sent across all channels. An example forwarding application was developed to illustrate the benefits of movability. The application consists of three components which was run, separately, on a Cooja simulator [Com14] and a single T-mote Sky; the experiments were initially conducted using a single T-mote Sky but the serial connection used to retrieve the memory usage data was being corrupted for as-yet unknown reasons, so we present results obtained from the simulator (Figure 5.2(a)) and the T-mote Sky (Figure 5.2(b)) (the river cats experiment above was performed on the T-mote's only for direct comparison with [CHS13]). The sender generates an array of packets which are sent to a forwarding component which simply sends the packets on. The forwarder also pauses for a number of seconds before waiting for more packets to be sent from the sender. The pause is to simulate the assumed behaviour of a forwarder in a real application; either its own processing of the packets or calculation of some routing information. Figure 5.2(a) presents the results obtained from running the two versions of the forwarding application on the Cooja simulator. Figure 5.2(b) presents the results obtained from running the two versions of the forwarding application on a single T-mote Sky. As expected, the memory usage for the movable version is considerably lower than for the non-movable version (in both experiments) due to the deep-copying performed by the sender and forwarder in the non-movable version. The output from the movable version finishes earlier in the figures because the x-axis measures the number of malloc and free operations during the simulation run, and since the movable version uses less memory, this number is necessarily smaller. Note, in Figure 5.2(b) the anomaly at the beginning of execution; while it alters the output somewhat, the periodic behaviour starting after around 1000 allocations clearly shows the benefits of movability. While the forwarding example is contrived, it is not uncommon for an intermediary component(s) to be present in a particularly complex application. Indeed, we saw this very setup in the river cats program between the data logger, the sensor sampler, and the flash storage component. Additionally, an Insense version of SNEE [Gal+11] would benefit from similar uses of movability.

We compare the external fragmentation of the non-movable and movable version of the forwarding application running on the Cooja simulator in Figures 5.3(a) and 5.3(b), respectively. As can be seen, external fragmentation in the movable version of the application is less severe than in the non-movable version due to its reduced number of memory allocations. Similar improvements running on the T-mote Sky are achieved but because of anomalies in the output, as explained earlier, we omit these graphs.

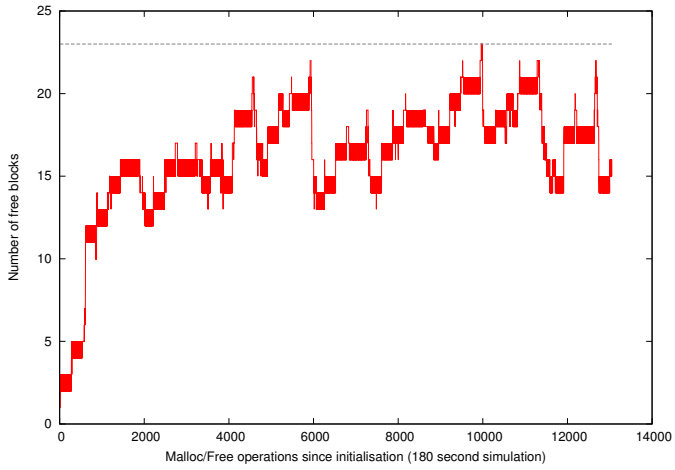


(a) Memory usage graph for the forwarding example program simulated in Cooja.

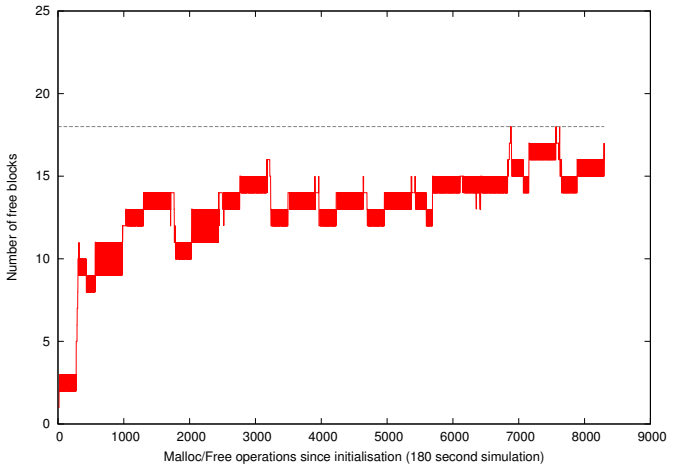


(b) Memory usage graph for the forwarding example program on physical T-Mote Sky.

Figure 5.2: Memory usage graph for the forwarding example program.

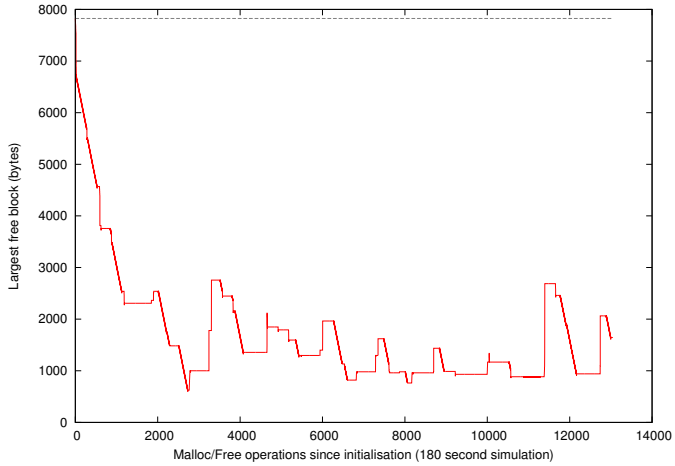


(a) External fragmentation for non-movable version of forwarding application.

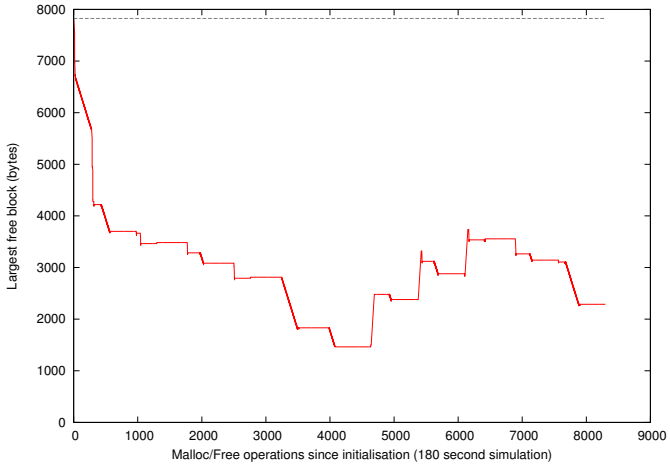


(b) External fragmentation for movable version of forwarding application.

Figure 5.3: External fragmentation for the forwarding example program.



(a) Largest free block over time non-movable version of forwarding application.



(b) Largest free block over time for movable version of forwarding application.

Figure 5.4: Largest free block over time for the forwarding example program.

We compare the largest free block over the simulation run of the non-movable and movable version of the forwarding application running on the Cooja simulator in Figures 5.4(a) and 5.4(b), respectively. As can be seen, the movable version has larger free blocks throughout its execution compared to the non-movable version. The reduced fragmentation and better utilisation of memory enables larger blocks to stay relatively untouched.

Chapter 6

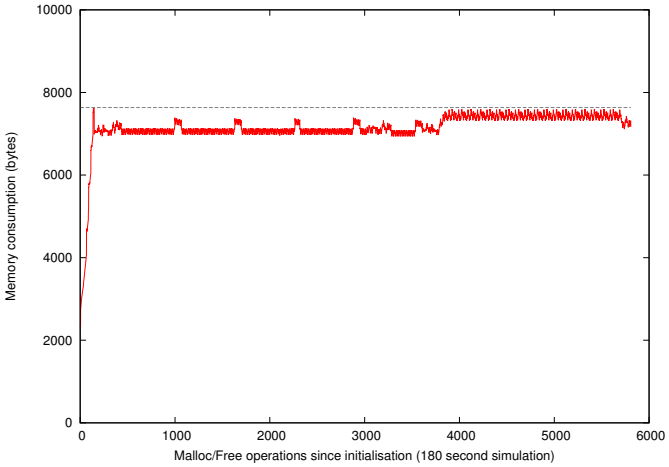
Conclusion and Future Work

We have presented an interprocedural data flow analysis to enforce soundness of a language extension used to improve the efficiency of the message passing mechanism within the Insense programming language. We have presented empirical results to show the improvements obtained from utilising movable memory, including overall memory consumption, less severe external fragmentation, and larger free blocks.

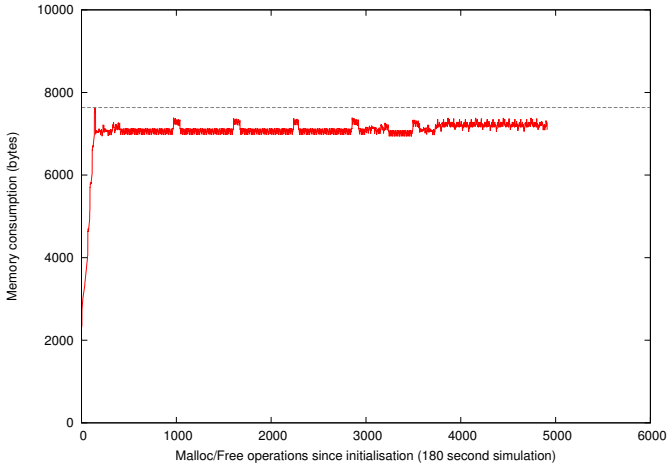
6.1 Future Work

The movability property is currently not supported for arrays, and channels contained within structures. Furthermore, Insense does not support nested structure types. Thus, the movability property for nested structures described in Section 3.1.1.1 was not implemented. Future work would be to extend the analysis to cope with the extra complexities involved in nesting references within structures, for example, handling overlapping alias regions.

Implementing these changes would improve the efficency savings obtained from the river cats application. We reproduce our experimental results for the river cats application from Chapter 5 (Figure 5.1(b)) in Figure 6.1(a) which shows the current savings due to movability changes which can be guaranteed sound by the compiler (statically enforced). Figure 6.1(b) illustrates further improvements possible once the described future work is implemented to guarantee soundness. Finally, Figure 6.1(c) highlights (in yellow) the changes to the source program to obtain the improvements shown in Figure 6.1(b) from the version of the program that produced Figure 6.1(a).



(a) Memory usage graph for the movable version of the River cats program.



(b) Memory usage graph for River cats program with future work movability extensions.

```
34  proc createReadStruct(in integer[] reader; integer[] dataBuffer) : readStruct
35   {
36       c = new out integer[];
37       connect c to reader;
38       rs = mov new readStruct(c, dataBuffer);
39       return rs;
40   }
41
42   proc createDataPacket(unsigned addr; unsigned id; integer[] buf) : RadioPacket
43   {
44       ds = new dataPacket(id, buf);
45       rp = mov new RadioPacket(addr, any(ds));
46       return rp;
47   }
```

(c) Source code change required to enable additional efficiency savings.

Figure 6.1: Comparing memory usage graphs for movable river cats programs.

Bibliography

[All70] Frances E. Allen. "Control Flow Analysis". In: *Proceedings of a Symposium on Compiler Optimization*. Urbana-Champaign, Illinois: ACM, 1970, pp. 1–19. DOI: 10.1145/800028.808479. URL: http://doi.acm.org/10.1145/800028.808479.

[Cal+90] David Callahan et al. "Constructing the Procedure Call Multigraph". In: *IEEE Trans. Softw. Eng.* 16.4 (Apr. 1990), pp. 483–487. ISSN: 0098-5589. DOI: 10.1109/32.54302. URL: http://dx.doi.org/10.1109/32.54302.

[CBC93] Jong-Deok Choi, Michael Burke, and Paul Carini. "Efficient Flow-sensitive Interprocedural Computation of Pointer-induced Aliases and Side Effects". In: *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '93. Charleston, South Carolina, USA: ACM, 1993, pp. 232–245. ISBN: 0-89791-560-7. DOI: 10.1145/158511.158639. URL: http://doi.acm.org/10.1145/158511.158639.

[CHS13] Callum Cameron, Paul Harvey, and Joseph Sventek. "A Virtual Machine for the Insense Language". In: *The 6th International Conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications*. Bologna, Italy, 2013, pp. 1–10.

[Com14] The Contiki Community. *Contiki: The Open Source OS for the Internet of Things*. http://www.contiki-os.org/index.html. [Online; Accessed 24th March 2014]. 2014.

[CPN98] David G. Clarke, John M. Potter, and James Noble. "Ownership Types for Flexible Alias Protection". In: *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '98. Vancouver, British Columbia, Canada: ACM, 1998, pp. 48–64. ISBN: 1-58113-005-8. DOI: 10.1145/286936.286947. URL: http://doi.acm.org/10.1145/286936.286947.

[Dea+08] Alan Dearle et al. "A Component-Based Model and Language for Wireless Sensor Network Applications". In: *Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference*. COMPSAC '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 1303–1308. ISBN: 978-0-7695-3262-2. DOI: 10.1109/COMPSAC.2008.151. URL: http://dx.doi.org/10.1109/COMPSAC.2008.151.

[FO76] Lloyd D. Fosdick and Leon J. Osterweil. "Data Flow Analysis in Software Reliability". In: *ACM Comput. Surv.* 8.3 (Sept. 1976), pp. 305–330. ISSN: 0360-0300. DOI: 10.1145/356674.356676. URL: http://doi.acm.org/10.1145/356674.356676.

[Gal+11] Ixent Galpin et al. "SNEE: a query processor for wireless sensor networks". English. In: *Distributed and Parallel Databases* 29.1-2 (2011), pp. 31–85. ISSN: 0926-8782. DOI: 10.1007/s10619-010-7074-3. URL: http://dx.doi.org/10.1007/s10619-010-7074-3.

[GBL02] Samuel Z. Guyer, Emery D. Berger, and Calvin Lin. *Detecting Errors with Configurable Whole-program Dataflow Analysis*. Tech. rep. 2002.

[HL07] Galen C. Hunt and James R. Larus. "Singularity: Rethinking the Software Stack". In: *SIGOPS Oper. Syst. Rev.* 41.2 (Apr. 2007), pp. 37–49. ISSN: 0163-5980. DOI: 10.1145/1243418.1243424. URL: http://doi.acm.org/10.1145/1243418.1243424.

[Hog91] John Hogg. “Islands: Aliasing Protection in Object-oriented Languages”. In: *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA ’91. Phoenix, Arizona, USA: ACM, 1991, pp. 271–285. ISBN: 0-201-55417-8. DOI: 10.1145/117954.117975. URL: http://doi.acm.org/10.1145/117954.117975.

[HU73] Matthew S. Hecht and Jeffrey D. Ullman. “Analysis of a Simple Algorithm for Global Data Flow Problems”. In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’73. Boston, Massachusetts: ACM, 1973, pp. 207–217. DOI: 10.1145/512927.512946. URL: http://doi.acm.org/10.1145/512927.512946.

[Hun+07] Galen Hunt et al. “Sealing OS Processes to Improve Dependability and Safety”. In: *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys ’07. Lisbon, Portugal: ACM, 2007, pp. 341–354. ISBN: 978-1-59593-636-3. DOI: 10.1145/1272996.1273032. URL: http://doi.acm.org/10.1145/1272996.1273032.

[Kil73] Gary A. Kildall. “A unified approach to global program optimization”. In: *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. POPL ’73. Boston, Massachusetts: ACM, 1973, pp. 194–206. DOI: 10.1145/512927.512945. URL: http://doi.acm.org/10.1145/512927.512945.

[KSK09] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. 1st. Boca Raton, FL, USA: CRC Press, Inc., 2009. ISBN: 0849328802, 9780849328800.

[KU76] John B. Kam and Jeffrey D. Ullman. “Global Data Flow Analysis and Iterative Algorithms”. In: *J. ACM* 23.1 (Jan. 1976), pp. 158–171. ISSN: 0004-5411. DOI: 10.1145/321921.321938. URL: http://doi.acm.org/10.1145/321921.321938.

[Moz14] Mozilla. *The Rust Reference Manual*. http://static.rust-lang.org/doc/0.9/rust.html. [Online; Accessed 17th Feburary 2014]. 2014.

[MP11] Luca Mottola and Gian Pietro Picco. “Programming wireless sensor networks: Fundamental concepts and state of the art”. In: *ACM Comput. Surv.* 43.3 (Apr. 2011), 19:1–19:51. ISSN: 0360-0300. DOI: 10.1145/1922649.1922656. URL: http://doi.acm.org/10.1145/1922649.1922656.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. ISBN: 1-55860-320-4.

[Mye81] Eugene M. Myers. “A precise inter-procedural data flow algorithm”. In: *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL ’81. Williamsburg, Virginia: ACM, 1981, pp. 219–230. ISBN: 0-89791-029-X. DOI: 10.1145/567532.567556. URL: http://doi.acm.org/10.1145/567532.567556.

[Nad+12] Karl Naden et al. “A type system for borrowing permissions”. In: *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL ’12. Philadelphia, PA, USA: ACM, 2012, pp. 557–570. ISBN: 978-1-4503-1083-3. DOI: 10.1145/2103656.2103722. URL: http://doi.acm.org/10.1145/2103656.2103722.

[Ryd79] B. G. Ryder. “Constructing the Call Graph of a Program”. In: *IEEE Trans. Softw. Eng.* 5.3 (May 1979), pp. 216–226. ISSN: 0098-5589. DOI: 10.1109/TSE.1979.234183. URL: http://dx.doi.org/10.1109/TSE.1979.234183.

[SP78] Micha Sharir and Amir Pnueli. *Two approaches to interprocedural data flow analysis*. Tech. rep. Report No. 002. 251 Mercer Street, New York, N.Y. 10012: Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 1978.

[SP81] Micha Sharir and Amir Pnueli. “Two approaches to interprocedural data flow analysis”. In: *Program Flow Analysis: Theory and Applications*. Ed. by Steven S. Muchnick and Neil D. Jones. Englewood Cliffs, NJ: Prentice-Hall, 1981. Chap. 7, pp. 189–234.

[SY86] R E Strom and S Yemini. “Typestate: A Programming Language Concept for Enhancing Software Reliability”. In: *IEEE Trans. Softw. Eng.* 12.1 (Jan. 1986), pp. 157–171. ISSN: 0098-5589. DOI: 10.1109/TSE.1986.6312929. URL: http://dx.doi.org/10.1109/TSE.1986.6312929.

[Wad90] Philip Wadler. “Linear types can change the world!” In: *IFIP TC 2 Working Conference on Programming Concepts and Methods*. Ed. by M. Broy and C. Jones. North Holland, 1990, pp. 347–359.

[Whe14] David A. Wheeler. *SLOCCount*. http://www.dwheeler.com/sloccount/. [Online; Accessed 6th March 2014]. 2014.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. “Constant Propagation with Conditional Branches”. In: *ACM Trans. Program. Lang. Syst.* 13.2 (Apr. 1991), pp. 181–210. ISSN: 0164-0925. DOI: 10.1145/103135.103136. URL: http://doi.acm.org/10.1145/103135.103136.