Universität des Saarlandes
Faculty of Natural Sciences and Technology I
Department of Computer Science

Bachelor's Thesis

An Intermediate Language To Formally Justify Memory Access Reordering

submitted by
Ralf Jung
on **27 March 2013**

Supervisor
Prof. Dr. Sebastian Hack

Advisors
Sigurd Schneider
Prof. Dr. Sebastian Hack

Reviewers
Prof. Dr. Sebastian Hack
Prof. Dr. Gert Smolka

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, ____________________ (Date) ____________________ (Signature)

Acknowledgement

I would like to thank my advisor Sigurd Schneider for many hours of discussion and encouraging me to just follow my ideas. His feedback and criticism has helped me profoundly to come up with the results of this thesis, and he gave me invaluable lessons in scientific writing.

Thanks also go to my supervisor Sebastian Hack for providing me with this challenging and interesting topic and his continuing feedback and support throughout my work on this thesis.

Furthermore, I would like to thank Sebastian Hack and Gert Smolka for reviewing my thesis.

Abstract

Typical intermediate representations used by compilers maintain a total order on memory operations. This means that memory access reordering is only done within an optimisation phase and must be explicitly combined with other analyses. We describe an intermediate language based on alias information which can represent absence of dependencies between memory operations. This enables all transformations to soundly reorder memory operations based solely on the structure of the program. The language is accompanied by a notion of well-typedness which supports proofs of correctness for transformations relaxing the dependencies between memory operations. Moreover, there always exists an easy translation to machine code for well-typed programs.

Contents

1 Introduction **1**
1.1 Contribution . . . 3
1.2 Outline . . . 3

2 Related Work **5**
2.1 Memory Representations . . . 5
2.2 Representing the Program as a Graph . . . 6
2.3 Memory Operations . . . 6
2.4 Reasoning about memories . . . 9

3 IL/M Syntax and Semantics **11**
3.1 Memory Operations . . . 12
3.2 Formal Semantics . . . 14
3.3 Realisability and Program Normalisation . . . 18

4 Type System for IL/M **21**
4.1 Informal Description . . . 22
4.2 Formalisation . . . 24

5 Properties of Well-Typed Programs **29**
5.1 Type Consistency and Preservation . . . 29
5.2 Semantic Equivalence and Normalisation . . . 34

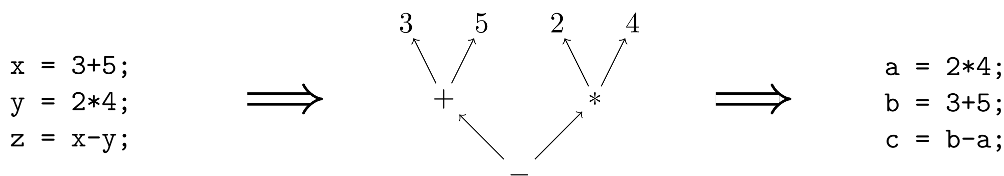
6 Evaluation **41**
6.1 Formalisation in Coq . . . 41
6.2 Reality Check . . . 43

7 Conclusion **45**
7.1 Limitations and Future Work . . . 46

Chapter 1

Introduction

Intermediate languages are used to decouple optimisations and code generation from the source language. They abstract away from details which are not needed for compilation, while typically having much more structure than the target language. For example, intermediate languages can discard the precise order in which computations were originally written down:



The graph preserves all the relevant information – which operations are performed on which operands. Variable names and the original order of execution are forgotten. Of course, the operands must be evaluated before the operations using them. Instead of the original total order, the graph defines a partial order. To convert the graph back to a textual program, we can choose any total order that respects the partial order. All the programs we obtain this way have the same behaviour as the original program. Therefore, reordering operations on which the graph inflicts no order is inherent to the representation: No further justification is needed to prove that semantics are preserved. Similar graphs can be used to represent entire programs, as described by Click in his idea of a 'sea of nodes' [CP95].

However, this approach does not work well for memory access. Without further knowledge concerning the values of pointers, memory operations must not be reordered. Two subsequent **store** operations on the *same* location must keep their order, while two **store** operations on *different* locations can be reordered.

In these program snippets, the value of `v` is written to the memory location denoted by `a`, and the value of `w` is written to the memory cell referenced by `b`.

In case we do not know whether `a` and `b` take the same value, it makes a difference which **`store`** is performed later. As the statements must not be swapped, it is important to remember their original order in the intermediate representation, for example:

If, however, we assume that `a` and `b` never take the same value when the program is run, the two **`store`** operations of the example are independent of each other. We may then use the same graph for both programs, thus making reordering of the operations inherent to the language.

In this thesis we present an intermediate language called IL/M which can express independence of memory accesses. IL/M is able to represent a subset of C without compound types or pointer arithmetic. Notably, it does not enforce memory safety, i.e., it can model memory access to invalid locations. It is an extension of IL/F by Schneider [Sch13], an intermediate language designed to justify the translation between low-level register transfer languages and high-level functional languages.

To obtain the necessary information about pointer values, we assume that we are given the results of an alias analysis [Cou86; SWH09, Sect. 1.11; RL12]. Such an analysis computes relations on values of pointer variables (namely, equality and/or inequality) which hold for each program execution. That information provides the justification to reorder memory operations, so we encode it in IL/M such that it remains available.

To ensure that program behaviour is left unchanged, we present a type system for IL/M enforcing the presence of alias information which is needed to justify the independence of memory operations. As long as the resulting program is well-typed, adding or removing memory dependencies does not change program behaviour. This can be used by a transformation which relaxes the order on memory operations to justify that it preserves program semantics. Two memory operations which are not ordered can be swapped without further justification. This transformation is inherent to the language, similar to how independent arithmetic computations can be conducted in arbitrary order.

Abstracting away from unnecessary details of the source language makes it easier to perform analyses and optimisations. This is one of the major reasons why intermediate languages are used in compilers. We hope that by relaxing the order of memory accesses where possible, the compiler gains more opportunity for optimisation. Besides, the compiler ultimately has to choose a total order for the program to emit machine code. During that phase, having a partial order on memory accesses allows for more choice and thus more optimisation.

The work of this thesis is formalised and verified using the Coq theorem prover.[1] We follow the convention that theorems, lemmas and corollaries presented in this thesis were proven in Coq, while conjectures just come with the proof sketch given here.

[1]We used Coq 8.4, which can be found at `http://coq.inria.fr/`.

1.1 Contribution

We describe IL/M, a language based on functional stores with inherent support for reordering independent memory operations. Our language can be used for source languages which are not memory safe, for example C.

Furthermore, we present a type system for IL/M which supports program transformations that add or remove memory dependencies. The type system is based on alias information annotated at the program. We show that two well-typed programs performing the same operations, but having different dependencies between memory operations, are semantically equivalent. The type system enforces the presence of dependencies between memory accesses unless their independence can be derived from the alias information. This property also ensures that the program can be easily translated to machine code, despite the use of functional stores.

To the best of our knowledge, no type system using alias information to justify independence of memory operations and realisability of functional stores on real machines has been described before.

1.2 Outline

Chapter 2 covers related work we built on to design IL/M, most notably graph-based program representations, different memory models and operations affecting memory.

In Chapter 3, we present IL/M, the language used throughout the thesis. We formalise the semantics of the language and explain why it can model the memory behaviour of C.

Chapter 4 describes the type system on an intuitive level and explains the safety guarantees it provides. Then we give the formal definition of well-typedness.

In Chapter 5, we discuss properties of well-typed programs. This includes type safety, and the fact that two well-typed programs differing only in the organisation of their memory dependencies are semantically equivalent.

Finally, Chapter 6 presents the formalisation of IL/M and the theorems we presented in Coq. We evaluate whether the language can fulfil its promises based on a real-world alias analysis.

Chapter 2

Related Work

In this chapter, we discuss previous work concerning memory representations and operations. We explain how graph-based program representations express memory dependencies. To relax such dependencies, an operation which subdivides a memory in two parts can be used. This requires reasoning about properties of memories, hence we introduce separation logic and capabilities, two approaches to formalise this reasoning.

2.1 Memory Representations

Typical intermediate representations for imperative source languages [ASU86, Chap. 8; Muc97, Chap. 4] rely on an implicit memory. This means that the memory is never explicitly mentioned in the source code. Such a language is also used by CompCert [Ler09b; Ler09a], a complete formal verification of a realistic C compiler in Coq.

```
store(a, v);
store(b, w);
```

Using implicit memory has the advantage of being close to the behaviour of actual machines, and also often close to how memory operations are handled in the source language. The translation from source to intermediate language is simple, and it is well understood that the semantics is preserved.

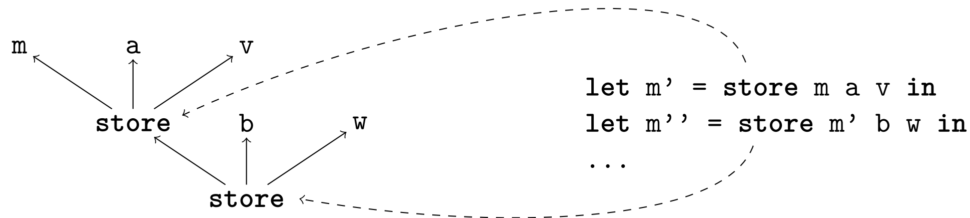
An alternative is to use functional stores [Str00, Sect. 3.3.2]: The memory is an explicit, immutable mapping from locations to values. Memory operations consume and produce such objects just like arithmetic operations consume and produce integers. The result of a **store** operation is a new memory, used by the next memory operation. Above example could be written as follows (using functional notation now, since we are dealing with a functional approach):

```
let m' = store m a v in
let m'' = store m' b w in
...
```

2.2 Representing the Program as a Graph

Graph-based program representations like the control flow graph [Pro59; All70] or the program dependence web [OBM90] are often employed to facilitate dataflow analyses and optimisations.

Another example, which we will discuss briefly in the following, is the 'Sea of Nodes' [CP95], which has been implemented by the FIRM intermediate representation [BBZ11]. The idea is to represent operations of the program using nodes and let edges denote operands. Recall the initial example for functional stores:



Variables which are free in the program (not bound by any `let`) also appear as nodes in the graph. Edges point to where the operand values come from, which is either a previous operation or a free variable. This makes the flow of data visible. In the graph, one can easily see that the second `store` makes use of the result of the first one. If imperative memory would be used, this dependency would not be immediately visible. An extra edge would have to be added to ensure that the two `store` are ordered properly.

Since the effect of each operation is completely described by its operands, no semantic information is lost when translating from a program to a graph. It follows immediately that two programs corresponding to the same graph, i.e., two different linearisations respecting the partial order induced by the graph, are semantically equivalent. This equivalence is inherent to the language and hence easy to justify by just looking at the program – no further analysis is required.

2.3 Memory Operations

In the following, we discuss the memory operations which are used by FIRM as well as those presented by Steensgaard [Ste95]. Both describe operations on functional stores corresponding to constructs usually found in imperative source languages. Table 2.1 on the facing page shows roughly corresponding operations in the two approaches. Furthermore, Steensgaard introduces memory operations which do not have a counterpart in common source or machine languages. They are used to express the dependencies between memory accesses more precisely. Neither FIRM nor Steensgaard's approach come with formal semantics of these operations.

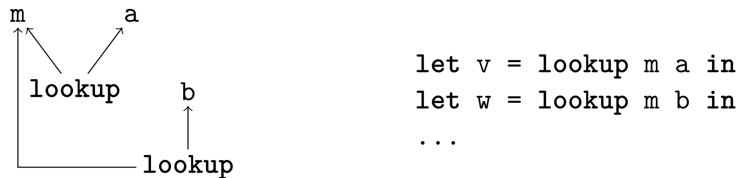
Steensgaard	FIRM
update	**store**
lookup	**load**
build	**alloc**
	free

Table 2.1: Roughly corresponding operations in FIRM and Steensgaard's approach

2.3.1 Common Operations: Steensgaard

The **update** operation is used to alter memory contents. It requires a memory, a location and a value as parameters, returning a new memory where the given location was updated.

Retrieving values is done by the **lookup** operation. It takes a memory and a location, and returns the value at the given location. Such a representation has the advantage that the order of multiple **lookup** operations in the same memory is undefined in the graph and hence reordering them is easily possible. For these two **lookup**, the order of execution obviously does not matter:




```
let v = lookup m a in
let w = lookup m b in
...
```

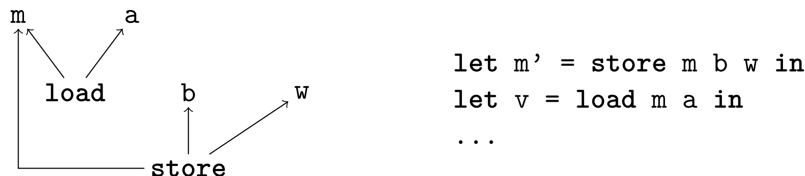
build creates new memories. It is given the location for which the memory is to be created and the initial value and returns a singleton memory. As there is no memory argument, several **build** following each other are not ordered in the graph.

```
let m1 = build a v in
let m2 = build b w in
...
```

Steensgaard describes no operation to dispose memory locations.

2.3.2 Common Operations: FIRM

The **store** operation in FIRM behaves exactly like Steensgaard's **update**. For **load** however, the situation is different. It returns not only a value, but also a memory which is identical to the one it got as argument and to be used for future operations. The reason for this is that **load** operations should not be moved up or down too far in the graph, namely beyond the next memory-modifying operation:




```
let m' = store m b w in
let v = load m a in
...
```

The program on the right is a correct linearisation of the graph. The two operations could be swapped without changing semantics, which is clear as the graph inflicts no order on them. However, a naïve translation of the given linearisation to real machine code will perform main memory accesses for **`store`** and **`load`**, ignoring the memory argument. This translation is incorrect. The **`load`** must return the value at location `a` in memory `m`, however, after the translation it loads from `m'` instead as `m` is no longer available. Real machines have only one memory to work on. Hence FIRM serialises **`load`** operations.

Allocating memory is done using **`alloc`**. It takes a memory as an argument and returns both a location, and a new memory – the old one extended by the new location. This can be used to introduce a strict ordering of allocation operations. However, it is also possible to pass an empty memory, thereby relaxing the order of subsequent **`alloc`** operations similar to how **`build`** behaves.

Furthermore, FIRM provides an operation **`free`** to remove a location from a memory. It is very similar to **`store`**. Instead of taking a value as an argument, it marks the given location as unallocated and returns the altered memory.

2.3.3 Operations to Reduce Dependencies

Steensgaard [Ste95] presents two more operations which manipulate a functional memory. These operations are used solely to structure the dependencies of memory operations.

A memory can be restricted to a part of its domain, and two disjoint memories (which do not both assign a value to the same location) can be merged. Using these operations, the strong linearisation of memory operations can be broken up and later, if necessary, merged again. This results in a structure as presented on the left in Figure 2.1. Since the two **`update`** are not ordered by the graph, it is easy to justify that they can be reordered. Two **`restrict`** operations are needed since only disjoint memories can be used with a **`merge`**. It is unclear which memory should have precedence if both overlap.

A similar structure can be expressed in FIRM as described by Mallon [Mal08, Sect. 4.1.4]: The total order on memory operations is broken up by independent operations using the same memory object, followed by a **`sync`**, which produces a memory used as

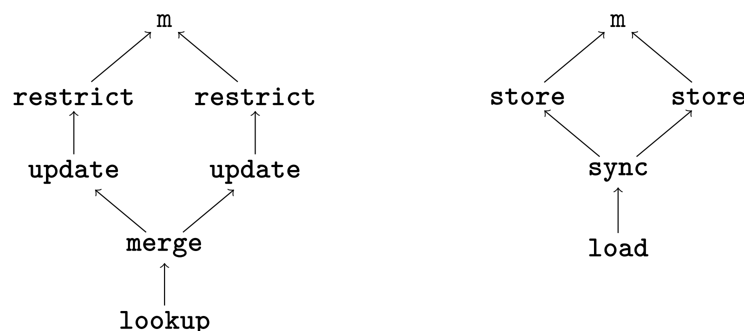


Figure 2.1: Two approaches to represent absence of memory dependencies
Left: Steensgaard's **`restrict`**, Right: **`sync`** in FIRM as described by Mallon

input for the next operation. The purpose of `sync` is to enforce subsequent operations to be executed after its arguments. This is shown in the second graph, where the `load` must be executed after both `store` happened.

2.4 Reasoning about memories

To obtain suited semantics for memory operations, we need a formal description of how to subdivide a memory in two parts. This requires reasoning about the domain of the memory, and specifying how the locations in this domain are separated to obtain two disjoint memories.

2.4.1 Separation Logic

Separation Logic [Rey02] has the purpose of making assertions about memories (to be used, e.g., as an extension of Hoare logic [Hoa69]). The central idea is the *separating conjunction* $\phi * \psi$ stating that ϕ and ψ apply to *disjoint parts* of the memory.

At first glance, it may seem that separation-logical formulae could serve as description of memory domains. A separating conjunction partitions the memory, fitting well to the concept of splitting it in two parts.

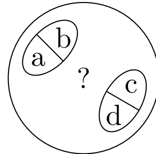
However, the separating conjunction abstracts away from how the memory is separated. There could be several ways to subdivide a memory in order to show that a separating conjunction holds. This is an important and often useful feature of separation logic, however, as a result semantics of an operation using the separating conjunction to split a memory would not be deterministic.

To avoid non-determinism, we need to be able to compute the domain of the split-off memory and describe it as separation-logical formula. If, for example, we want to split `a` and `b` into a new memory, we need a formula describing a memory with only `a` and `b` in its domain. Depending on whether they are equal or not, the memory contains one or two elements. This could be described by $(\mathtt{a} \mapsto - * \mathtt{b} \mapsto -) \vee (\mathtt{a} \mapsto - \wedge \mathtt{b} \mapsto -)$. Here, $\mathtt{a} \mapsto -$ denotes a memory which contains exactly `a`, i.e., it describes a singleton memory. The left part describes the case that `a` and `b` are not equal; a memory satisfying this formula will always have exactly two elements. The right-hand side describes the case of both being equal: A single memory is both a singleton memory containing only `a`, and a singleton memory for `b`.

As one sees, this approach does not scale well. We have to enumerate all cases of aliasing and non-aliasing of variables we have no information about, while the information we want to convey is that the memory contains exactly `a` and `b` and nothing else.

We believe the reason for this problem is a fundamental difference in the overall structure of separation logic and alias information. Separation logic is tailored for a top-down view on the memory. It describes how the memory is organized, separating it into smaller and smaller pieces:

Alias information has a very different structure. It is necessarily incomplete [Ram94] and local. We might know that `a` and `b` are different, and so are `c` and `d`. We have no idea how `a` and `c` or any of the other pairs relate:



Enumerating all these small local disjoint memories embeds the alias information into separation logic. However, using separation logic to represent alias information adds significant overhead.

2.4.2 Capabilities

The calculus of capabilities [WCM00] is a type system where memory operations require *capabilities* to be performed. By using affine environments [Gir87] (each value must be used at most once), aliasing is restricted. Well-formed programs under this calculus can never perform invalid memory operations, like accessing freed or unallocated memory or freeing a memory twice.

However, this approach is not suited for memory-unsafe languages. The goal of the calculus of capabilities is to provide guarantees about program behaviour. A compiler for a language like C cannot come up with such guarantees. The compiler must be able to compile C programs which *can* go wrong, and preserve their semantics for all the cases in which they do not.

Based on the idea of capabilities, Charguéraud and Pottier [CP08] developed an imperative language with implicit memory which can be transformed to a purely functional language using an explicit memory representation. The design goal here was to make reasoning about imperative programs easier by controlling ownership of memory regions using types. The capabilities, which are just part of the type system without any semantics in the imperative language, are translated to maps representing memories. The capability types also provide a separating conjunction. Operations on capabilities include focusing and unfocusing a variable from and back into a group, which is equivalent to splitting and merging a single variable from/into a memory.

The issues of the calculus of capabilities concerning unsafe languages apply here as well. In addition, a run-time system with a garbage collector is assumed, which makes the language unfit for a C compiler.

Chapter 3

IL/M Syntax and Semantics

We based our work on IL/F [Sch13], a simple first-order functional language. Only tail calls are possible, i.e., the return value of the called function is also the return value of the caller. That makes the language easier to translate to machine code. IL/M extends IL/F with basic memory operations as described by the syntax in Figure 3.1 below.

Here, f, x, a and m range over variables. By convention, f denotes a function, x a value, a describes a location and m a memory. Furthermore, $\overline{x}$ ranges over lists of variables, A over finite sets of variables, and e over expressions. Note that we do not specify the structure of expressions – that simplifies the formalisation and keeps it general enough to support arbitrary operators. However, we assume expressions to operate on variables only (no memory access or function application) and to have no side-effects.

The focus of this language is on memory operations, and being able to express how they depend on each other. We need further information to justify the independence of

$s, t ::= \{G\}\ \texttt{let}\ x = e\ \texttt{in}\ s$	variable binding
$\mid \{G\}\ \texttt{if}\ x\ \texttt{then}\ s\ \texttt{else}\ t$	conditional
$\mid \{G\}\ x$	function return
$\mid \{G\}\ \texttt{fun}\ f\ \overline{x}\ m = s\ \texttt{in}\ t$	function definition
$\mid \{G\}\ f\ \overline{x}\ m$	function application
$\mid \{G\}\ \texttt{let}\ m = \texttt{store}\ m\ a\ x\ \texttt{in}\ s$	memory store
$\mid \{G\}\ \texttt{let}\ x = \texttt{load}\ m\ a\ \texttt{in}\ s$	memory load
$\mid \{G\}\ \texttt{let}\ m, a = \texttt{alloc}\ \texttt{in}\ s$	memory allocation
$\mid \{G\}\ \texttt{let}\ m = \texttt{free}\ m\ a\ \texttt{in}\ s$	memory deallocation
$\mid \{G\}\ \texttt{let}\ m, m = \texttt{split}\ m\ A\ \texttt{in}\ s$	splitting memory
$\mid \{G\}\ \texttt{let}\ m = \texttt{merge}\ m\ m\ \texttt{in}\ s$	merging memories

Figure 3.1: Syntax of IL/M

memory operations. This information is encoded as annotation for each statement: G ranges over the possible annotations, which we will detail later. Given a statement s, we write G_s to denote the information annotated at s. We will omit the annotation if it does not matter.

In the remainder of this chapter, we discuss the memory operations of IL/M and we present their formal semantics. We also explain why these semantics are suited to model the behaviour of C when it comes to memory access.

3.1 Memory Operations

The design goal of IL/M is to express independence of memory operations in the program itself. We consider operations independent if the program graph imposes no order on them. Using multiple memory variables, the order becomes partial, so that we have a wide choice of linearisations which are semantically equivalent.

This would not be possible using implicit memory: To ensure that program behaviour is completely preserved, memory operations must be entirely linearised in the graph. A relaxation of this order cannot be expressed in the language.

The formal treatment in this thesis works on the textual representation of the program. Such representations are much easier to define inductively than graphs, which makes them better suited for a formalisation in Coq. Inductive program definitions also lend themselves very well to inductive definitions of semantics and (as discussed in Chapter 4) well-typedness of programs using inference rules. Nevertheless, it is often helpful to keep the program's corresponding graph in mind, so we will usually give both representations in the following, sometimes using a simplified graph to keep the focus on the important dependencies.

Our memory operations are taken both from Firm [BBZ11] and the work by Steensgaard [Ste95], as shown in Table 3.1. See Section 2.3 for a detailed discussion of these operations.

The `alloc` of IL/M behaves like its Firm counterpart with an empty memory as argument. It will choose a fresh location and return this location plus a singleton memory containing an uninitialised cell at the given location. This models the behaviour of `malloc` in C which does not initialise the memory area it provides.

IL/M operation	taken from (original name)
`store`	Firm (**`store`**)
`load`	Steensgaard (**`lookup`**)
`alloc`	similar to Firm (**`alloc`**)
`free`	Firm (**`alloc`**)
`split`	similar to Steensgaard (**`restrict`**)
`merge`	Steensgaard (**`merge`**)

Table 3.1: Sources for memory operations in IL/M

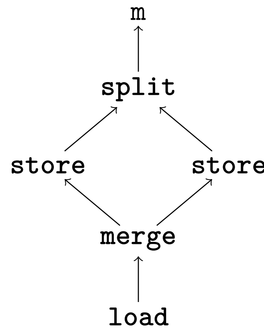
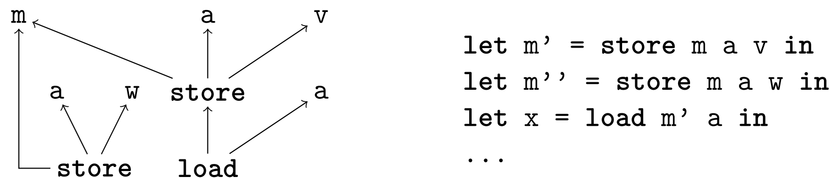


Figure 3.2: Our approach to represent absence of memory dependencies

Instead of the **restrict** operation described by Steensgaard, we use an operation called **split**. It essentially performs two **restrict** operations at once. **split** takes a set of variables as argument. Two new memories are returned, one containing the given subset of the locations of the original memory, the other one containing the remaining locations. We did not use existing frameworks like separation logic [Rey02] or capabilities [WCM00] here for the reasons described in Section 2.4. **split** is visualised in Figure 3.2 (compare this to Figure 2.1 on page 8). For reasons we will discuss in Chapter 4, this approach is more suited for our needs than **restrict** or **sync**.

3.1.1 Realisability

The use of functional stores induces a new problem: This language is further away from how an actual machine works. It can describe programs which cannot be directly compiled to real machine code. Consider the following example.



The first **store** creates a new memory `m'`. It differs from the memory referenced by `m` only in the value at the location denoted by `a`, which in `m'` is changed to `v`. The second **store** creates another new memory object `m''`, independent of `m'`, where `a` is mapped to the value of `w`. The final **load** is performed on `m'`, hence it will always return `v`.

However, a naïve translation of this program, which compiles each memory operation to direct memory access, is incorrect. Both **store** operations would be performed on the machine memory, therefore the final **load** would return `w`. The fact that **load** explicitly referenced `m'` as memory to load from is lost during translation, since only one memory is available on an actual machine. The simulation behaves as if the final **load** was performed on `m''`.

We say that a linearised program permitting such a naïve translation is *realisable*. This notion is formalised later in Section 3.3. The example program above is not realisable.

The issue of realisability has already been touched when we discussed the reason for `load` in FIRM to return a memory (see Section 2.3.2). The example we used there is a graph which has both realisable and non-realisable linearisations. FIRM tries to avoid such graphs so that every linearisation can be chosen to emit machine code.

3.2 Formal Semantics

To formally reason about program behaviour, we define small-step semantics for IL/M as shown in Figure 3.3 on page 16. There are a few definitions and notations we need to formulate the inference rules. These are presented below.

Definition 1 (Environment)**.** An *environment* Γ for a given set V is a function *variables* $\to \{\bot\} \cup V$. The *empty environment* $\emptyset$ maps every variable to $\bot$. The *domain* $\operatorname{dom}\Gamma$ of an environment is the set of variables not mapped to $\bot$. An environment Γ_1 is a *sub-environment* of Γ_2, in symbols $\Gamma_1 \subseteq \Gamma_2$, if both assign the same value to each variable in $\operatorname{dom}\Gamma_1$ (but Γ_2 may have a larger domain).

The notation Γ^a_v describes the environment obtained by updating Γ to return v for variable a:

$$(\Gamma^a_v)\ x = \begin{cases} v & \text{if } x = a \\ \Gamma\ x & \text{otherwise} \end{cases}$$

Lookup and update can also be performed using lists $\overline{x}$ of variables (for update, two lists of equal length are required).

Definition 2 (Memory)**.** A *memory* μ is a function *values* $\to \{\bot, \square\} \cup$ *values* mapping locations to memory cells. Here, $\bot$ denotes a cell which is unallocated, while $\square$ represents an uninitialised cell: an allocated cell no value has been written to yet. The *empty memory* $\emptyset$ maps each location to $\bot$.

The domain of a memory and the notation to update it are defined similarly to environments. Note that uninitialised cells are contained in the domain.

Definition 3 (Closure and closure context)**.** A *closure* $c = (E, s, \overline{x}, m)$ contains the value environment E it closes over, the statement s to execute, the list $\overline{x}$ of formal variable parameters and the name m of the formal memory parameter. We will use the notation $c.E$ to refer to the value environment of a given closure c, and similar for the other components.

A *closure context* L is a list of named closures. $\emptyset$ denotes the empty context, and L^f_c describes the context obtained by adding closure c with name f to context L. Using a name twice is not allowed, so this is only valid if no closure called f is already contained in the list. To look up a name, we use the notation $L\ f$ denoting the (unique) closure with name f. If no such closure exists, we have $L\ f = \bot$.

The *domain* $\operatorname{dom} L$ of a closure context is the set of names added to it. The notation $L|_{f..}$ describes the context obtained by dropping everything declared after f (excluding f itself) from L.

We do not use environments of closures for the bookkeeping to avoid having a function environment in the closures. We can use $L|_{f..}$ to obtain the context f was declared in. This keeps the formal description closer to our Coq formalisation, which is described in Section 6.1.

Definition 4 (State)**.** A *state* $R = (E, L, S, U \mid s)$ consists of an environment E for values, a closure context L, an environment S for memories, the set U of locations which have been allocated so far, and the statement s which is to be executed. Again, $R.E$ etc. refers to the components of a given state R.

We will often have a fixed program s and want to talk about arbitrary states for that program. Then we use $Q = (E, L, S, U)$ to denote the values, memories, closures and used addresses, and $(Q \mid s)$ describes the entire state. $Q.L$ etc. can be used to refer to the components of Q.

Similar to expressions, we do not define the structure of values. Furthermore, we do not formally differentiate between values and locations – any value can be used as a location. However, the type system will later impose some restrictions. We still speak of locations to make explicit the intention that this value is used to address a cell in a memory.

Semantics are given as transition relation between states. The judgement $R_1 \to R_2$ denotes that a single step takes the first state to the second one. As usual $\to^*$ is the reflexive transitive closure of $\to$, i.e., $R_1 \to^* R_2$ means the second state can be reached from the first by any number of steps (including zero). We designed the semantics so that it is *deterministic*, i.e., given two successors R' and R'' of the same state ($R \to R'$ and $R \to R''$), we have $R' = R''$.

The inference rules are presented in Figure 3.3 on the following page. They show some specifics of our approach which we discuss in the following. Note that we omitted the annotation $\{G\}$ for each statement since it does not influence the semantics.

Exp uses the judgement $E \vdash e \Downarrow v$ for expression evaluation: Expression e evaluates in value environment E to value v. We further assume for If that we are provided with a function $\mathit{val2bool} : \mathit{values} \to \{0, 1\}$ to use any value in a conditional. The precise details of evaluation and converting to boolean can easily be filled in when our approach is used for a concrete implementation.

Fun and App create and use closures. Application of a function is performed in the closure context $L|_{f..}$ obtained by dropping everything declared after f, which is the context at the time when f was declared, plus f itself. Note that the closure does not contain any memory, it only closes over the value environment E. We want the simulation of IL/M on an actual machine to be straight-forward. However, putting memory into the closure would require the simulation to restore this memory on application of the function. Instead, the memory must be passed as an argument, which is easy to implement on a real machine – memory is simply left unchanged when jumping to the callee.

We do not even allow for passing several memories to a function: Everything must be merged back to one memory. The rationale behind this is that moving statements across function application is non-trivial anyway. This transformation changes the graph

$$\textsc{Exp}\ \frac{E \vdash e \Downarrow v}{E, L, S, U \mid \texttt{let}\ x = e\ \texttt{in}\ s \ \ \rightarrow\ \ E^x_v, L, S, U \mid s}$$

$$\textsc{If}\ \frac{\mathit{val2bool}(E\,x) = i}{E, L, S, U \mid \texttt{if}\ x\ \texttt{then}\ s_1\ \texttt{else}\ s_0 \ \ \rightarrow\ \ E, L, S, U \mid s_i}$$

$$\textsc{Fun}\ \frac{c = (E, s, \overline{x}, m)}{E, L, S, U \mid \texttt{fun}\ f\ \overline{x}\ m = s\ \texttt{in}\ t \ \ \rightarrow\ \ E, L^f_c, S, U \mid t}$$

$$\textsc{App}\ \frac{L\,f = (E', s, \overline{x}, m) \qquad L' = L|_{f..}}{E, L, S, U \mid f\ \overline{y}\ m' \ \ \rightarrow\ \ E'^{\,\overline{x}}_{E\,\overline{y}}, L', \emptyset^m_{S\,m'}, U \mid s}$$

$$\textsc{Store}\ \frac{(S\,m)(E\,a) \neq \bot \qquad \mu = (S\,m)^{E\,a}_{E\,x}}{E, L, S, U \mid \texttt{let}\ m' = \texttt{store}\ m\ a\ x\ \texttt{in}\ s \ \ \rightarrow\ \ E, L, S^{m'}_{\mu}, U \mid s}$$

$$\textsc{Load}\ \frac{(S\,m)(E\,a) = v \qquad \bot \neq v \neq \square}{E, L, S, U \mid \texttt{let}\ x = \texttt{load}\ m\ a\ \texttt{in}\ s \ \ \rightarrow\ \ E^x_v, L, S, U \mid s}$$

$$\textsc{Alloc}\ \frac{v = \mathit{allocate}\ U \qquad \mu = \boldsymbol{\emptyset}^v_{\square}}{E, L, S, U \mid \texttt{let}\ m, a = \texttt{alloc}\ \texttt{in}\ s \ \ \rightarrow\ \ E^a_v, L, S^m_\mu, U \cup \{v\} \mid s}$$

$$\textsc{Free}\ \frac{(S\,m)(E\,a) \neq \bot \qquad \mu = (S\,m)^{E\,a}_{\bot}}{E, L, S, U \mid \texttt{let}\ m' = \texttt{free}\ m\ a\ \texttt{in}\ s \ \ \rightarrow\ \ E, L, S^{m'}_{\mu}, U \mid s}$$

$$\textsc{Split}\ \frac{\mu' = (S\,m)|_{E\,A} \qquad \mu'' = (S\,m)|_{\mathrm{dom}(S\,m) \setminus E\,A}}{E, L, S, U \mid \texttt{let}\ m', m'' = \texttt{split}\ m\ A\ \texttt{in}\ s \ \ \rightarrow\ \ E, L, S^{m'\ m''}_{\mu'\ \mu''}, U \mid s}$$

$$\textsc{Merge}\ \frac{\mathrm{dom}(S\,m) \cap \mathrm{dom}(S\,m') = \emptyset \qquad \mu = S\,m \sqcup S\,m'}{E, L, S, U \mid \texttt{let}\ m'' = \texttt{merge}\ m\ m'\ \texttt{in}\ s \ \ \rightarrow\ \ E, L, S^{m''}_{\mu}, U \mid s}$$

Figure 3.3: Small-step semantics of IL/M

$$\textsc{Big-Return}\ \frac{}{E, L, S, U \mid x \Downarrow E\,x}$$

$$\textsc{Big-Step}\ \frac{R \rightarrow R' \qquad R' \Downarrow v}{R \Downarrow v}$$

Figure 3.4: Big-step semantics of IL/M

corresponding to the program, hence it requires further reasoning about correctness. Allowing arbitrary memories to be passed around would severely complicate handling function application in the type system presented later. Hence we decided to enforce merging all memories before applying a function and concentrate on supporting program transformations which do not change the program graph.

Store and Load require the location they operate on to be allocated and initialised, respectively. If this is not the case, the program gets stuck. This models undefined behaviour of the C source program – see also the discussion in Section 3.2.1.

Free is very similar to store, the only difference is that, instead of writing a value to the memory location, it writes $\bot$ to mark that location as unallocated. The location is however not removed from U – there can still be pointers to this location, so we do not want `alloc` to allocate a new memory at that location. As we will discuss later, that would significantly complicate handling memory in the type system.

Alloc uses a function *allocate* to determine the address of a fresh memory cell. We require that function to depend on U alone and to return a value which is not contained in U, but we do not rely on the actual behaviour of the allocator. If we allowed using any $v \notin U$ in Alloc, semantics would not be deterministic.

Split and Merge constitute no surprises. Note that Merge forces the memories to be disjoint. They use the following operations on memory objects:

Definition 5 (Restriction of a memory)**.** The *restriction* of a memory μ to a set of values A, in symbols $\mu|_A$, is defined as

$$(\mu|_A)\, x = \begin{cases} \mu\, x & \text{if } x \in A \\ \bot & \text{otherwise} \end{cases}$$

Definition 6 (Union of memories)**.** The *union* $\mu_1 \sqcup \mu_2$ of two memories μ_1 and μ_2 is defined as

$$(\mu_1 \sqcup \mu_2)\, x = \begin{cases} \mu_1\, x & \text{if } \mu_2\, x = \bot \\ \mu_2\, x & \text{otherwise} \end{cases}$$

Note that this operation is only commutative if the domains of μ_1 and μ_2 are disjoint.

Defining big-step semantics based on these inference rules is straight-forward, see Figure 3.4 on the preceding page. The judgement $(Q \mid s) \Downarrow v$ says that program s terminates when starting it in the given environments and returns v. Obviously, this is equivalent to $(Q \mid s) \to^* (Q' \mid x)$ with $Q'.E\, x = v$.

3.2.1 Relation to the C Standard

The goal of our work is to create a memory model powerful enough to simulate the C memory semantics. Therefore, design decisions are driven by the requirements of the C specification – in the remainder of this section, we refer to the C11 ISO standard [C11]. We must ensure that IL/M programs can only get stuck if their behaviour in C is undefined. Otherwise, translating from C to IL/M would not be semantically correct.

When a memory location is unallocated, both **load** and **store** operations on that cell get stuck. The C standard allows this, since a pointer to such a cell can actually never be dereferenced: It could be obtained using pointer arithmetic, but that is only allowed within an array object (Section 6.5.6) and only if the result is actually still contained in the array, or at the address right after it – but in the latter case, dereferencing that pointer is not allowed. After calling the function `free`, pointers to unallocated objects may still be available, but they may not be dereferenced either (Section 6.5.3.3).

In addition, **load** gets stuck when accessing an uninitialised memory location. The value of an object at such a location is indeterminate (Section 7.22.3.4), which means that it can be a trap representation (Section 3.19.2). Reading such a trap representation results in undefined behaviour unless a character pointer is used (Section 6.2.6.1). Therefore we disallow reading it. A full C implementation would have to allow reading it using a character pointer, which however would require an explicit notion of an indeterminate value in the program semantics, which we tried to avoid to keep semantics simple and deterministic.

In our semantics, **alloc** will never return the same pointer twice. Every location returned by **alloc** is added to U, and it is never removed again, not even by **free**. The C standard actually does not even allow a program to observe whether the implementations of `malloc` and `free` re-use memory locations. After calling `free`, the values of pointers to the freed object become indeterminate (Section 6.2.4) and cannot be compared to pointers returned by future calls to `malloc`. Therefore, the application has no way to even find out whether `malloc` re-uses a pointer or not. IL/M currently does not make use of this freedom, pointers can be compared even after the location they reference has been freed. They cannot be dereferenced any more, however.

3.3 Realisability and Program Normalisation

In order to give a formal definition of realisability, we introduce the notion of *program normalisation*. Normalisation is a transformation which changes the program to use only one memory variable m. All **split** and **merge** operations are removed, the memory argument of all **load**, **store** and **free** operations is ignored and replaced by m. Each **alloc** is immediately followed by a **merge** to merge the new singleton memory into m.

This is defined formally in Figure 3.5 on the facing page: $norm_m\ s$ is the normalised form of s using m as name for the global memory. $\hat{m}$ is the name of a temporary memory used for allocation; it must be different from m. For example, when using natural numbers as representation of variable names, we can set $\hat{m} := m + 1$.

Based on the definitions of this chapter, we can now make a first attempt to formalise the notion of realisability introduced in Section 3.1.1:

Definition 7 (Realisability, preliminary)**.** A program s is *realisable* if the following holds for some m:

$$\forall Q, v : (Q \mid s \Downarrow v) \Rightarrow (Q \mid norm_m\ s \Downarrow v)$$

$$
\begin{aligned}
&norm_m(\{G\}\,\texttt{let}\,x = e\,\texttt{in}\,s) &&= \{G\}\,\texttt{let}\,x = e\,\texttt{in}\,norm_m\,s\\
&norm_m(\{G\}\,\texttt{if}\,x\,\texttt{then}\,s\,\texttt{else}\,t) &&= \{G\}\,\texttt{if}\,x\,\texttt{then}\,norm_m\,s\,\texttt{else}\,norm_m\,t\\
&norm_m(\{G\}\,x) &&= \{G\}\,x\\
&norm_m(\{G\}\,\texttt{fun}\,f\,\overline{x}\,m' = s\,\texttt{in}\,t) &&= \{G\}\,\texttt{fun}\,f\,\overline{x}\,m = norm_m\,s\,\texttt{in}\,norm_m\,t\\
&norm_m(\{G\}\,f\,\overline{x}\,m') &&= \{G\}\,f\,\overline{x}\,m\\
&norm_m(\{G\}\,\texttt{let}\,m'' = \texttt{store}\,m'\,a\,x\,\texttt{in}\,s) &&= \{G\}\,\texttt{let}\,m = \texttt{store}\,m\,a\,x\,\texttt{in}\,norm_m\,s\\
&norm_m(\{G\}\,\texttt{let}\,x = \texttt{load}\;m'\,a\,\texttt{in}\,s) &&= \{G\}\,\texttt{let}\,x = \texttt{load}\;m\,a\,\texttt{in}\,norm_m\,s\\
&norm_m(\{G\}\,\texttt{let}\,m', a = \texttt{alloc}\,\texttt{in}\,s) &&= \{G\}\,\texttt{let}\,\hat{m}, a = \texttt{alloc}\,\texttt{in}\\
& &&\quad \{G_s\}\,\texttt{let}\,m = \texttt{merge}\,\hat{m}\,m\,\texttt{in}\,norm_m\,s\\
&norm_m(\{G\}\,\texttt{let}\,m'' = \texttt{free}\,m'\,a\,\texttt{in}\,s) &&= \{G\}\,\texttt{let}\,m = \texttt{free}\,m\,a\,\texttt{in}\,norm_m\,s\\
&norm_m(\{G\}\,\texttt{let}\,m'', m''' = \texttt{split}\,m'\,A\,\texttt{in}\,s) &&= norm_m\,s\\
&norm_m(\{G\}\,\texttt{let}\,m''' = \texttt{merge}\,m'\,m''\,\texttt{in}\,s) &&= norm_m\,s
\end{aligned}
$$

Figure 3.5: Program normalisation

In other words: If the program terminates with result v in the initial environments Q, the normalised program using global memory m must behave the same. That program in turn can easily be simulated on an actual machine, using the real memory to simulate m.

This definition does not take well-typedness and consistency of states into account, which is why we will change it later (see Section 5.2). However, it already gives the idea.

Looking at the initial example for non-realisability from Section 3.1, one can easily see that normalisation changes semantics. The left-hand side shows the original program, the right-hand side is normalised for `m`:

```
let m' = store m a v in          let m = store m a v in
let m'' = store m a w in         let m = store m a w in
let x = load m' a in             let x = load m a in
...                              ...
```

The left-hand program will always bind `x` to the value of `v`. The right-hand side behaves as if the `load` was performed on `m''` since the normalisation forgets which variable was used, so `x` will be bound to the value of `w`. This is exactly what would happen if the left-hand side would be naïvely translated to machine code.

Concerning computability, we conjecture that realisability of a program is undecidable.

Conjecture 1. *Realisability of a program s is not decidable.*

Proof sketch. Realisability can be reduced to the halting problem. Assume there is a deciding procedure for realisability. Now, given some program s which is structurally equal to its normalisation, we know that s is realisable. Append the following non-realisable program fragment (based on above example) to s:

```
let m' = store m a v in
let m'' = store m a w in
let x = load m' a in x
```

If s does not terminate, then it is still realisable as the left-hand side of the implication in Definition 7 is always false. If however s terminates, it is not realisable since normalisation changes semantics: s will return `v` while $norm_m\ s$ returns `w`.

Since the halting problem is undecidable, this reduction implies that realisability is undecidable as well.

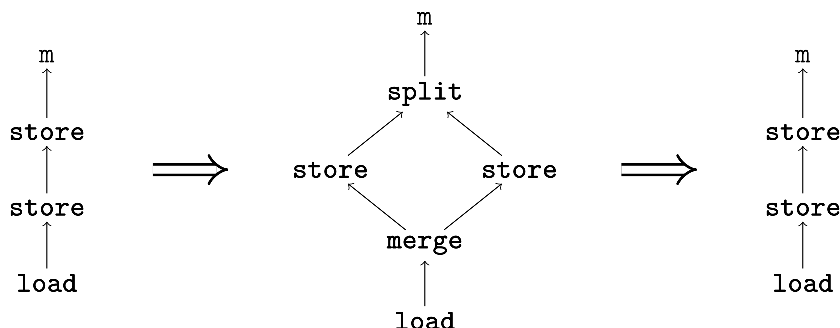
Chapter 4

Type System for IL/M

In this chapter we describe a type system for IL/M. It is designed in such a way that well-typed programs are guaranteed to be realisable, i.e., easy to translate to a real machine. Furthermore, it provides support for introducing **split** and **merge** operations into a program, a process we call *de-linearisation*, based on pointer equalities and inequalities annotated at the program. To achieve this, memory variables are equipped with types describing their domain.

The central property of the type system is: If a program s is well-typed, then its normalisation $norm_m\ s$ (as defined in Figure 3.5 on page 19) is well-typed as well, and both are semantically equivalent. This relies on the alias annotation being correct.

A compiler workflow using IL/M could work as follows: The source program is translated to IL/M with entirely linearised memory operations, as the source language usually does not allow for anything else. Based on the well-typedness of the original program, the compiler can prove the IL/M program to be well-typed. Using information obtained, e.g., by an alias analysis, the compiler then de-linearises memory dependencies. The details of this transformation, i.e., how to find out where **split** and **merge** operations should be inserted, are beyond the scope of this thesis. After proving that the resulting program is well-typed, above property guarantees that if normalising the transformed program yields the original one, both are semantically equivalent.



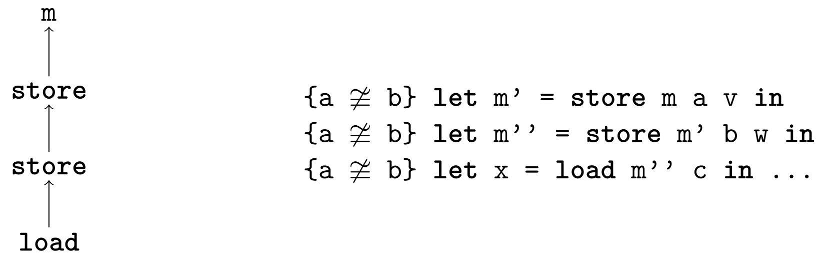
The subsequent optimisation phases are performed on the de-linearised program, preserving semantics and well-typedness. Finally, the compiler has to linearise the last

program again to translate it to machine code. Since that program is well-typed, the type system again guarantees that it is semantically equivalent to its normalised form.

During the remainder of this chapter, we first give an idea on how this is achieved, then we present the formalisation of the type system. Properties of well-typed programs, like the aforementioned semantic equivalence, are covered in Chapter 5.

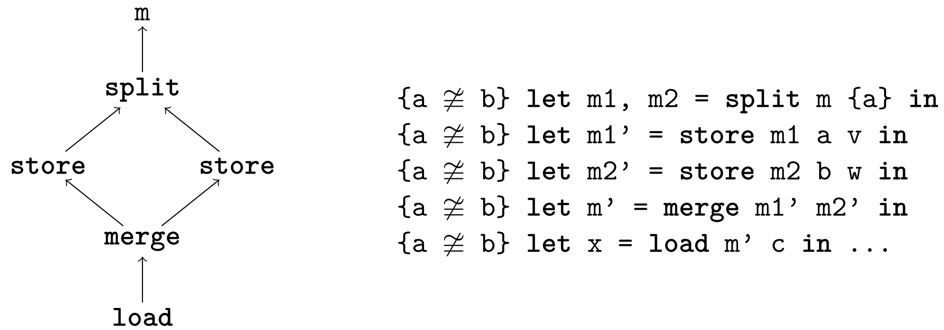
4.1 Informal Description

Consider the following simple IL/M program:



Without further information about the values of `a` and `b`, there is not much we can do. Hence we assume some analysis found out that these pointers never take the same value. We do not use the inequality sign here since `a` $\not\cong$ `b` is not a statement about the variable names `a` and `b`. It rather is a statement about the *values* of `a` and `b` for each execution of the program.

Now we want to express, in the structure of the program itself, that this makes the two **`store`** operations independent of each other. This is achieved by introducing **`split`** and **`merge`** operations:



It is important here that we merged the memory before performing the **`load`**: Since we have no information whatsoever regarding the value of `c`, both **`store`** operations could have an influence on the result of **`load`**. Hence we ensure that the **`load`** is ordered below both **`store`** in the graph.

The type system keeps track of which variables were **`split`** to a separate memory. These variables are called the *focus*. The memory which contains the split-off locations (in this case, this is only the location denoted by `a`) is called a *focus memory*. The type

of such a memory is the set of variables which was used to create it, so that the set of corresponding values describes its domain. Therefore, the focus is always the union of the types of all focus memories.

All locations which have not been split into a focus memory remain in the *panorama memory*. This accounts for the fact that there are always locations a compiler does not know anything about. Real-world alias information is necessarily incomplete [Ram94]. Hence the panorama memory collects all these locations. For obvious reasons, there always has to be exactly one panorama memory, and it has the type $\top$.

The following table shows, for each statement of the example program, the memory type environment and the focus before the statement. For the sake of readability, we use an empty table cell instead of $\bot$ for undeclared variables.

`m`	`m1`	`m2`	`m1'`	`m2'`	`m'`	focus	
$\top$						`{}`	`{a ≇ b} let m1, m2 = split m {a} in`
	`{a}`	$\top$				`{a}`	`{a ≇ b} let m1' = store m1 a v in`
		$\top$	`{a}`			`{a}`	`{a ≇ b} let m2' = store m2 b w in`
			`{a}`	$\top$		`{a}`	`{a ≇ b} let m' = merge m1' m2' in`
					$\top$	`{}`	`{a ≇ b} let x = load m' c in ...`

To justify this transformation, we have to prove the new program to be semantically equivalent to the old one. The type system imposes some requirements to be able to provide this equivalence: When accessing a location through **`store`** or **`load`**, we require the variable to be *accessible* in the memory used for that operation. Accessibility is defined depending on the type of the memory: For a focus memory, the variable has to be equal to one of the variables contained in the memory type. To access the panorama memory, the variable must have a different value than *all* variables in the focus. These proofs of (in)equality must be derived from the alias annotations at the statement in question. Only if the required relation is *statically derivable* from the alias information, memory access is allowed. Based on a correctness proof of the alias analysis we can then show that the (in)equalities hold for each execution and make use of them in proofs concerning the semantics of an IL/M program.

In our example, the following arguments justify accessibility:

1. The first **`store`** uses `a` to access memory `m1` which has type `{a}`. Since `a` $\cong$ `a` is trivial, accessibility is easily shown.
2. The second **`store`** uses `b` to access the panorama memory while the focus is `{a}`. To prove accessibility, we have to show `a` $\ncong$ `b`, which is given by the annotation.
3. Finally, the **`load`** uses `c` to access the panorama memory while the focus is empty. Here, there is nothing to show.

As you can see, the rules for accessibility require the inequality `a` $\ncong$ `b` to be present, which is exactly the inequality justifying the independence of the two **`store`** operations. In other words, without this inequality, the transformed program would not have been well-typed.

In addition, accessibility actually requires us to merge memories before performing the **load**: c cannot be used to access m1' since we cannot prove it to be equal to a. It cannot be used to access m2' either since it cannot be proven to be unequal to a. As explained above, this is important to preserve the semantics of the program. In general, access to variables with insufficient information is prohibited.

Note that accessibility does not guarantee that the memory access will actually succeed: It could still be the case that the location is not allocated. However, a location accessible in a memory can never be found in another memory.

We also have to ensure that the transformed program is still realisable. This is done by enforcing the domains of all memories, which can be used at a program point, to be disjoint. As you can see in the table above, m has no type any more after the **split**: It is removed from the type environment. This prevents m from being used again later, which would violate realisability. Similarly, **store**, **free** and **merge** also remove their input variables from the type environment. Intuitively, memory domains being disjoint guarantees that the compiler never has to duplicate parts of the memory to simulate the behaviour of IL/M on a real machine. This explains why **restrict** and **sync** as presented in Section 2.3.3 are not suited for our approach. In both cases, we would have to allow non-disjoint memories being available at the same time.

Types which make sure variables are used only once are called *affine types*. If, furthermore, variables may not be discarded (i.e. they must be used exactly once), the types are *linear* [Gir87; Wad93]. That is however not needed in our case. The type system we present is partially affine: The single exception is **load**, which allows the memory to be used again afterwards.

4.2 Formalisation

As already mentioned, variable equality and inequality as used by accessibility is based on the alias information annotated at the statement in question. A proof of accessibility requires a proof that the required equalities and inequalities can be derived from the information assumed to be valid at this program point. Provability is defined by a proof system on alias annotations which is given in Figure 4.1 on the facing page. The alias annotations are finite sets of *alias tokens* κ. Such a token has the form $a \ncong b$ or $a \cong b$ where a and b are variables. The judgement $G \rhd \kappa$ says that the validity of token κ can be derived from the set of given tokens G.

A memory type ϕ is either a set A of variables (then ϕ is a focus type) or $\top$, which denotes the panorama memory. Based on these rules, it is straight-forward to define accessibility.

Definition 8. A variable a is *accessible* in a memory of type ϕ under alias information G and focus F, in short $G, F \rhd a \mathrel{\tilde{\in}} \phi$, if the following holds:

- If $\phi = \top$, then a must not alias with any focus variable: $\forall a' \in F : G \rhd a \ncong a'$
- Otherwise, $\phi = A$ is a set of variables and a must alias with some variable in A: $\exists a' \in A : G \rhd a \cong a'$

$$\text{ASSUMPTION}\ \frac{\kappa \in G}{G \rhd \kappa}$$

$$\text{EQ}_{\text{REFL}}\ \frac{}{G \rhd a \cong a} \qquad \text{EQ}_{\text{SYMM}}\ \frac{G \rhd a_2 \cong a_1}{G \rhd a_1 \cong a_2}$$

$$\text{EQ}_{\text{TRANS}}\ \frac{G \rhd a_1 \cong a_3 \qquad G \rhd a_3 \cong a_2}{G \rhd a_1 \cong a_2}$$

$$\text{NEQ}_{\text{SYMM}}\ \frac{G \rhd a_2 \ncong a_1}{G \rhd a_1 \ncong a_2} \qquad \text{NEQ}_{\text{EQ}}\ \frac{G \rhd a_1 \cong a_3 \qquad G \rhd a_3 \ncong a_2}{G \rhd a_1 \ncong a_2}$$

Figure 4.1: Derivation rules for alias tokens

We will also need the following notion of two sets of variables which will have disjoint sets of values for every execution:

Definition 9 (Alias-disjointness)**.** Two sets of variables A_1 and A_2 are *alias-disjoint* under alias information G, in short $G \rhd A_1 \boxdot A_2$, if the following holds:

$$\forall a_1 \in A_1, a_2 \in A_2 : G \rhd a_1 \ncong a_2$$

Similar to the state in dynamic semantics, the static context collects the information about variables which is needed to judge whether a program is well-typed.

Definition 10 (Closure type and closure type context)**.** A *closure type* $\chi = (\Gamma, \overline{\theta}, \tau)$ consists of the type environment Γ for the values in the closure, the list of argument types $\overline{\theta}$ and the return type τ. The notation $\chi.\Gamma$ describes the type environment of a given closure type χ, and similar for the other components.

A *closure type context* Λ is a list of named closure types. We use the same notations as we did for closure contexts (see Definition 3 on page 14).

Definition 11 (Static context)**.** A *static context* $(\Gamma, \Lambda, \Sigma)$ consist of an environment Γ for the types of normal variables (holding values), a close type context Λ and an environment Σ for memory types.

The *focus* F_Σ of a memory type environment Σ is defined as the union of all focus memory types:

$$F_\Sigma := \bigcup_{m \in \operatorname{dom} \Sigma,\ \Sigma\, m \neq \top} \Sigma\, m$$

The type inference rules for common operations are given in Figure 4.2 on the following page. The judgement $\Gamma, \Lambda, \Sigma \vdash s : \tau$ says that program s is well-typed with return type τ in the static context $(\Gamma, \Lambda, \Sigma)$.

Similar to how we dealt with values and expressions, we leave the definition of types up to the implementation. We assume that for every type τ there is a type $^*\tau$ for pointers to values of type τ. Furthermore, we assume a judgement $\Gamma \vdash e : \tau$ saying that expression e has type τ in environment Γ, and a judgement $\vdash v : \tau$ stating that value v is

$$\text{T-Exp}\ \frac{\Gamma\, x = \bot \qquad \Gamma \vdash e : \tau' \qquad \neg \mathit{isPtr}(\tau') \qquad G \subseteq G_s \qquad \Gamma^{x}_{\tau'}, \Lambda, \Sigma \vdash s : \tau}{\Gamma, \Lambda, \Sigma \vdash \{G\}\ \texttt{let}\ x = e\ \texttt{in}\ s : \tau}$$

$$\text{T-If}\ \frac{\Gamma\, x = \tau' \qquad G \subseteq G_s \qquad \Gamma, \Lambda, \Sigma \vdash s : \tau \qquad G \subseteq G_t \qquad \Gamma, \Lambda, \Sigma \vdash t : \tau}{\Gamma, \Lambda, \Sigma \vdash \{G\}\ \texttt{if}\ x\ \texttt{then}\ s\ \texttt{else}\ t : \tau}$$

$$\text{T-Ret}\ \frac{\Gamma\, x = \tau}{\Gamma, \Lambda, \Sigma \vdash \{G\}\ x : \tau}$$

$$\text{T-Fun}\ \frac{\begin{array}{cccc} \Lambda\, f = \bot \qquad c = (\Gamma', \overline{\theta}, \tau') & \Gamma' \subseteq \Gamma & \Gamma'\,\frac{\overline{x}}{\overline{\theta}}, \Lambda^{f}_{c}, \emptyset^{m}_{\top} \vdash s : \tau' \\ G \subseteq G_t & & \Gamma, \Lambda^{f}_{c}, \Sigma \vdash t : \tau \end{array}}{\Gamma, \Lambda, \Sigma \vdash \{G\}\ \texttt{fun}\ f\ \overline{x}\ m = s\ \texttt{in}\ t : \tau}$$

$$\text{T-App}\ \frac{\Lambda\, f = (\Gamma', \overline{\theta}, \tau) \qquad \Gamma\, \overline{y} = \overline{\theta} \qquad \Sigma\, m' = \top \qquad F_{\Sigma} = \emptyset}{\Gamma, \Lambda, \Sigma \vdash \{G\}\ f\ \overline{y}\ m' : \tau}$$

Figure 4.2: Inference rules for well-typedness of IL/M programs: Common operations

of type τ. We define *isPtr* to be a predicate describing whether a given type is a pointer: $\mathit{isPtr}(\tau) :\Leftrightarrow \exists \tau' : \tau = {}^*\tau'$.

As you can see, the inference rules require the alias information to be monotone: The set of alias tokens can only grow, no token may be removed. The only exception is the body of a function, which is also the only place where variables can leave scope.

T-Exp does not allow a variable to be overwritten. The name must not have been in use before. This is not actually a restriction (variables can simply be renamed), and it avoids problems with variables changing their value while they are used to describe the domain of a memory. Furthermore, T-Exp requires the type τ' of the expression not to be a pointer type. This forbids pointer arithmetic. It also prevents simple assignment of pointers, which however can be easily avoided by always referring to the original variable.

Since functions can only get a single memory argument, T-Fun requires the function body to type-check in a memory type environment with just a panorama memory. Moreover, functions may not be overwritten as this is not allowed in closure (type) contexts. An arbitrary sub-environment Γ' of the currently existing variables Γ can be carried over to the closure.

T-App requires the focus to be empty before calling a function. This does not entirely forbid focus memories to exist, since memories of type $\emptyset$ can be obtained by splitting the empty set off a memory. However, it ensures that all existing memory locations are passed on to the function.

Figure 4.3 on the next page presents the inference rules for memory operations. We forbid overwriting memory variables to avoid 'loosing' locations or the panorama memory. However, memory variables are removed anyway after they were used and modified, to make memory types affine. Hence `let m = store m a x` is allowed. This is reflected by the typing rules requiring the memory variable to be untyped (i.e. have type $\bot$) after removing the soon-to-be-gone argument memory variables from the environment.

$$\textsc{T-Store}\ \dfrac{\begin{array}{c}(\Sigma^m_\bot)\, m' = \bot \qquad \Sigma\, m = \phi \qquad \Gamma\, x = \tau' \qquad \Gamma\, a = {}^*\tau' \\ G, F_\Sigma \rhd a \,\tilde{\in}\, \phi \qquad G \subseteq G_s \qquad \Gamma, \Lambda, \Sigma^{m\ m'}_{\bot\ \phi} \vdash s : \tau\end{array}}{\Gamma, \Lambda, \Sigma \vdash \{G\}\ \mathtt{let}\ m' = \mathtt{store}\ m\ a\ x\ \mathtt{in}\ s : \tau}$$

$$\textsc{T-Load}\ \dfrac{\begin{array}{c}\Gamma\, x = \bot \qquad \Sigma\, m = \phi \qquad \Gamma\, a = {}^*\tau' \qquad G, F_\Sigma \rhd a \,\tilde{\in}\, \phi \\ G \subseteq G_s \qquad \Gamma^x_{\tau'}, \Lambda, \Sigma \vdash s : \tau\end{array}}{\Gamma, \Lambda, \Sigma \vdash \{G\}\ \mathtt{let}\ x = \mathtt{load}\ m\ a\ \mathtt{in}\ s : \tau}$$

$$\textsc{T-Alloc}\ \dfrac{\begin{array}{c}\Gamma\, a = \bot \qquad \Sigma\, m = \bot \qquad \forall a' \in F_\Sigma : G' \vdash a \ncong a' \\ G \subseteq G_s \qquad \Gamma^a_{*\tau'}, \Lambda, \Sigma^m_{\{a\}} \vdash s : \tau\end{array}}{\Gamma, \Lambda, \Sigma \vdash \{G\}\ \mathtt{let}\ m, a = \mathtt{alloc}\ \mathtt{in}\ s : \tau}$$

$$\textsc{T-Free}\ \dfrac{\begin{array}{c}(\Sigma^m_\bot)\, m' = \bot \qquad \Sigma\, m = \phi \qquad \Gamma\, a = {}^*\tau' \\ G, F_\Sigma \rhd a \,\tilde{\in}\, \phi \qquad G \subseteq G_s \qquad \Gamma, \Lambda, \Sigma^{m\ m'}_{\bot\ \phi} \vdash s : \tau\end{array}}{\Gamma, \Lambda, \Sigma \vdash \{G\}\ \mathtt{let}\ m' = \mathtt{free}\ m\ a\ \mathtt{in}\ s : \tau}$$

$$\textsc{T-Split}_{\textsc{f}}\ \dfrac{\begin{array}{c}(\Sigma^m_\bot)\, m' = \bot \qquad (\Sigma^m_\bot)\, m'' = \bot \qquad \Sigma\, m = A \neq \top \qquad m' \neq m'' \\ G \rhd A' \boxdot (A \setminus A') \qquad A' \subseteq A \qquad G = G_s \qquad \Gamma, \Lambda, \Sigma^{m\ m'\ m''}_{\bot\ A'\ A \setminus A'} \vdash s : \tau\end{array}}{\Gamma, \Lambda, \Sigma \vdash \{G\}\ \mathtt{let}\ m', m'' = \mathtt{split}\ m\ A'\ \mathtt{in}\ s : \tau}$$

$$\textsc{T-Split}_{\textsc{p}}\ \dfrac{\begin{array}{c}(\Sigma^m_\bot)\, m' = \bot \qquad (\Sigma^m_\bot)\, m'' = \bot \qquad \Sigma\, m = \top \qquad m' \neq m'' \\ \forall a \in A : \mathit{isPtr}(\Gamma\, a) \qquad G \rhd A \boxdot F_\Sigma \qquad G = G_s \qquad \Gamma, \Lambda, \Sigma^{m\ m'\ m''}_{\bot\ A\ \top} \vdash s : \tau\end{array}}{\Gamma, \Lambda, \Sigma \vdash \{G\}\ \mathtt{let}\ m', m'' = \mathtt{split}\ m\ A\ \mathtt{in}\ s : \tau}$$

$$\textsc{T-Merge}_{\textsc{f}}\ \dfrac{\begin{array}{c}(\Sigma^{m\ m'}_{\bot\ \bot})\, m'' = \bot \qquad \Sigma\, m = A \neq \top \qquad \Sigma\, m' = A' \neq \top \\ m \neq m' \qquad G = G_s \qquad \Gamma, \Lambda, \Sigma^{m\ m'\ m''}_{\bot\ \bot\ A \cup A'} \vdash s : \tau\end{array}}{\Gamma, \Lambda, \Sigma \vdash \{G\}\ m'' = \mathtt{merge}\ m\ m'\ \mathtt{in}\ s : \tau}$$

$$\textsc{T-Merge}_{\textsc{p}}\ \dfrac{\begin{array}{c}(\Sigma^{m\ m'}_{\bot\ \bot})\, m'' = \bot \qquad \Sigma\, m = A \neq \top \qquad \Sigma\, m' = \top \\ G = G_s \qquad \Gamma, \Lambda, \Sigma^{m\ m'\ m''}_{\bot\ \bot\ \top} \vdash s : \tau\end{array}}{\Gamma, \Lambda, \Sigma \vdash \{G\}\ m'' = \mathtt{merge}\ m\ m'\ \mathtt{in}\ s : \tau}$$

Figure 4.3: Inference rules for well-typedness of IL/M programs: Memory operations

T-Store, T-Free and T-Load require the location to be accessible in the memory, as discussed above. Furthermore, the pointer used to access the location must of course have a pointer type, and the stored/loaded value has the corresponding type. Again, T-Free is very similar to T-Store.

T-Alloc creates a new focus memory. To maintain the consistency relation presented in Section 5.1, we also require some alias information about the new location to be present: After the allocation, the alias tokens must express the fact that the new location is different from every location which previously was in the focus. This is trivial, since *allocate* returns fresh locations, so it should be easy to establish that fact. Furthermore, during the initial translation to IL/M, the focus will usually be empty since **`split`** is not used and each **`alloc`** is immediately followed by a **`merge`**, so there will only be the panorama memory. In this case there is nothing to prove.

There are two variants of T-Split: One to work on the panorama memory, and one for focus memories. Splitting a focus memory requires a set of variables which is alias-disjoint from the remaining variables. Without that requirement, splitting {`a`} from {`a`, `b`} could also move `b` to the new memory, so we would not know how to type the remainder memory. When using **`split`** on the panorama memory, there is a similar requirement: The split-off variables must be alias-disjoint from the current focus. This is equivalent to requiring all variables to be accessible in the panorama memory. It guarantees that we do not add variables to the new focus memory which are actually already contained in another focus memory.

T-Merge also has two variants. The rules ensure that one does not merge a memory with itself. Note that only the second argument may be a panorama memory. Since there can only be one such memory, and **`merge`** is commutative, this is not a restriction.

Also, T-Merge and T-Split (in both variants) do not allow the alias information to change at all. There is no point in allowing information to be added here, and forbidding it is required for normalisation to preserve the annotation (see Lemma 4 on page 34).

An interesting question related to the type system is: Is well-typedness of a program in a given static context $(\Gamma, \Lambda, \Sigma)$ decidable? We conjecture that this is the case.

Conjecture 2. *Given a static context* $(\Gamma, \Lambda, \Sigma)$*, well-typedness of programs is decidable.*

Proof sketch. The structure of the program and the current Σ leave no choice about which inference rule to apply next, so this boils down to a deciding procedure for deriving alias tokens. We require the sets G of alias tokens to be finite, and we only deal with a finite set of variables, so it is possible to enumerate all equivalence classes of variables, where a and b are equivalent if $G \rhd a \cong b$. The fact that this is an equivalence relation follows from Eq_{REFL}, Eq_{SYM} and Eq_{TRANS} (see Figure 4.1 on page 25). As a result, $G \rhd a \cong b$ can be decided by checking whether a and b are in the same equivalence class. The rules NEq_{SYM} and NEq_{EQ} establish that $G \rhd a \ncong b$ can be viewed as a symmetric relation between these equivalence classes. For each inequality token in G, mark the equivalence classes as inequal to each other. Now $G \rhd a \ncong b$ can be decided by checking whether the equivalence classes of a and b are marked as unequal. Hence derivability of alias tokens is decidable. As a result, accessibility and well-typedness are decidable as well.

Chapter 5

Properties of Well-Typed Programs

Our type system establishes some guarantees on executions of well-typed programs. Based on a consistency relation between states and static contexts, *type safety* is separated into *progress* lemmas, proving that a state consistent with a context makes a step, and *preservation* lemmas, proving that if such a state makes a step, the next state is also consistent with a (possibly different) context [Pie02, Sect. 8.3]. Since programs in IL/M can get stuck to model undefined behaviour of the underlying C program, we cannot establish progress lemmas for all statements. However, IL/M ensures type preservation.

Furthermore, the type system provides a semantic guarantee: Well-typed programs are semantically equivalent to their normalisation (as defined in Figure 3.5 on page 19). This implies that well-typed programs are realisable, i.e., translating them to real machine code is straight forward (see Definition 7 on page 18).

5.1 Type Consistency and Preservation

Before discussing type preservation, we have to define which properties are to be preserved through executions of well-typed programs. Our consistency relation requires the values of variables in E to have the type described by Γ, and similar for functions. Memories must adhere to the domain specification given by their type. To have any typing guarantees about values obtained via **`load`**, we need an oracle σ (mapping locations to $\{\bot\} \cup$ *types*) telling us for each allocated memory location which type the content of that cell has. Then we can require memory cells to have values of that type, and pointers to point to cells with an appropriate type. This guarantees pointers point to values matching their type. A similar idea is used in proofs of type safety for ML, where the oracle is called *store typing* [Pie02, Sect. 13.4].

Since we can never have pointers with different types to the same location, such a static model of types of memory cells is sufficient. If we would allow pointer arithmetic or re-using the memory space released by **`free`** in a future **`alloc`**, we would need to include types in our run-time model of memory, similar to what CompCert does [LB08].

To formalise consistency between states and contexts, we need some preliminary definitions:

Definition 12 (Typing of value environments)**.** A value environment E *is typed according to* a value type environment Γ, in short $\vdash E : \Gamma$, if all values have the type given by Γ:

$$\forall a \in \operatorname{dom} \Gamma : \vdash E\ a : \Gamma\ a$$

Definition 13 (Typing of closure contexts)**.** A closure context L *is typed according to* a closure type context Λ, in short $\vdash L : \Lambda$, if for all closure names $f \in \operatorname{dom} \Lambda$ the closure $c = L\ f$ matches the closure type $\chi = \Lambda\ f$. The value environment $c.E$ must be typed according to the type environment $\chi.\Gamma$ in the closure type, and the statement $c.s$ has to be well-typed:

$$\vdash c.E : \chi.\Gamma \ \wedge \ \chi.\Gamma^{c.\overline{x}}_{\chi.\overline{\theta}}, \Lambda|_{f..}, \emptyset^{c.m}_{\top} \vdash c.s : \chi.\tau$$

Now we have all the formalism available to define the consistency of a state:

Definition 14 (Consistency of a state)**.** A state $(E, L, S, U \mid s)$ *is consistent with* the static context $(\Gamma, \Lambda, \Sigma)$ with return type τ using oracle σ if the following conditions hold. Consistency is denoted by the judgement $(E, L, S, U \mid s) : \tau \vDash_\sigma \Gamma, \Lambda, \Sigma$.

1. The memory type environment Σ is valid
   (a) $\forall a \in F_\Sigma : \mathit{isPtr}(\Gamma\ a)$ (the focus consists of pointers only)
   (b) $\exists! m^\top : \Sigma\ m^\top = \top$ (there is exactly one panorama memory)
   (c) $\forall m, m' \in \Sigma \setminus \{m^\top\} : m \neq m' \Rightarrow G_s \triangleright \Sigma\ m \boxdot \Sigma\ m'$ (the types of different focus memories are alias-disjoint)
2. $\Gamma, \Lambda, \Sigma \vdash s : \tau$ (the current statement is well-typed with type τ)
3. $\vdash E : \Gamma$ (the value environment is typed according to the static context)
4. $\vdash L : \Lambda$ (the label environment is typed according to the static context)
5. Memories are well-typed
   (a) $\forall m \in \operatorname{dom} \Sigma : S\ m \subseteq U$ (all memory domains are subsets of U)
   (b) $\operatorname{dom}(S\ m^\top) \cap E\ F_\Sigma = \emptyset$ (the panorama memory contains no focus location)
   (c) $\forall m \in \Sigma \setminus \{m^\top\} : \operatorname{dom}(S\ m) \subseteq E\ (\Sigma\ m)$ (the focus memories contain no more than the addresses denoted in their type)
6. The oracle fits to the state: Let μ be the memory obtained by merging all available memories, i.e., $\mu = \bigsqcup_{m \in \operatorname{dom} \Sigma} S\ m$ (as we will see later, all these memories are disjoint, so the order in which they are merged does not matter)
   (a) $\operatorname{dom} \sigma \subseteq U$ (the oracle only predicts types for allocated addresses)
   (b) $\forall v \in \operatorname{dom} \sigma : \bot \neq \mu\, v \neq \square \Rightarrow \vdash \mu\, v : \sigma\, v$ (the prediction of the oracle is correct for cells containing a value)

(c) $\forall a \in \operatorname{dom}\Gamma, \tau' : \Gamma\ a = {}^*\tau' \Rightarrow \sigma(E\ a) = \tau'$ (pointers in E point to cells with the right type)

(d) For all $f \in \operatorname{dom}\Lambda$, let $\chi = \Lambda\ f$ and $c = L\ f$; for all in $a \in \operatorname{dom}\chi.\Gamma$, we must have $\chi.\Gamma\ a = {}^*\tau' \Rightarrow \sigma(c.E\ a) = \tau$ (pointers in the value environments of the closures point to cells with the right type)

(e) $\forall v \in \operatorname{dom}\sigma, \tau' : \sigma\ v = {}^*\tau' \Rightarrow \sigma(\mu\ v) = \tau'$ (pointers in memory point to cells with the right type)

Note that the first condition involves only the static context. It excludes invalid contexts which otherwise allow well-typed programs, like memory environments Σ having several panorama memories, or memory types referring to non-existent variables.

The domain specifications of memory types can be conservative approximations of the reality. By condition 5c, the domain of a focus memory does not have to exactly match the variables in its type, it only has to be a subset thereof. As a result, there is no guarantee about the value of such a variable being contained in the memory. However, condition 5b ensures the value is *not* contained in the panorama memory.

To take any use of the annotated information, we need the alias tokens G to actually conform with the current values of variables. This is not a property well-typedness can maintain, hence it is not part of type consistency. We assume we are given correct alias information.

Definition 15 (Satisfying alias tokens, Alias-correctness)**.** A value environment E *satisfies* an alias token κ, in symbols $E \vDash \kappa$, if the following holds

- If $\kappa = a \cong b$, then $E\ a = E\ b$
- If $\kappa = a \ncong b$, then $E\ a \neq E\ b$

A value environment E *satisfies* a set G of alias tokens ($E \vDash G$) if E satisfies all tokens $\kappa \in G$.

An annotated program s which has type τ under the static context $(\Gamma, \Lambda, \Sigma)$ is *alias-correct* if all the states it reaches in a consistent execution have value environments satisfying the current annotation:

$$\begin{aligned} \forall \sigma, Q, R :\ & \Gamma, \Lambda, \Sigma \vdash_\sigma (Q \mid s) : \tau \\ & \Rightarrow Q \mid s \rightarrow^* R \\ & \Rightarrow R.E \vDash G_{R.s} \end{aligned}$$

An important property of the typing rules is that this consistency relation is maintained when a well-typed program performs a step from a state which satisfies the annotation:

Theorem 1 (Type preservation)**.** *Let R_1 be a state such that $R_1 : \tau \vDash_{\sigma_1} \Gamma_1, \Lambda_1, \Sigma_1$ and $R_1.E \vDash G_{R_1.s}$. Let R_2 be such that $R_1 \rightarrow R_2$. Then there exist Γ_2, Λ_2, Σ_2, σ_2 such that $R_2 : \tau \vDash_{\sigma_2} \Gamma_2, \Lambda_2, \Sigma_2$.*

The proof of this theorem by case distinction over the performed step – as actually conducted with the Coq theorem prover – is rather lengthy and cumbersome, so we present only the general ideas and the cases where preservation is not entirely obvious. Preserving consistency was the motivation for some of the requirements of the type inference rules which were already mentioned above.

If the value of a variable could change, conditions 5b and 5c (concerning the domains of memories) could be violated afterwards: The memories did not change, but the result of looking up the focus or the memory type, respectively. If the panorama memory could leave scope, condition 1b would be violated.

If functions were allowed arbitrary memory arguments, T-App would have to handle substitution in memory types to ensure that the domains of focus memories are still properly described by their types. Condition 6d is required to establish condition 6c again after performing a function application.

T-Split requires the split-off set to be alias-disjoint from the remainder or the focus, respectively, to ensure that the focus memory types remain alias-disjoint (condition 1c). Both T-Split and T-Merge choose the type of the resulting memory in a way that the domain restrictions (condition 5) are maintained.

`alloc`, **`free`** and **`store`** are the only operations affecting the merged memory μ, so maintaining condition 6 is trivial for all the others. Since T-Store checks the type of the pointer used to access the memory, condition 6e guarantees that the oracle is still valid after updating the memory. **`free`** just marks a cell as unallocated, so condition 6b is maintained without changing the oracle.

For **`alloc`**, the oracle is updated at $\mathit{allocate}\, U$ which (according to our assumptions about the *allocate* function) is not contained in U. Condition 6a guarantees that the oracle does not predict a type for this location, which in turn means that by conditions 6c–e, no pointer is pointing there. Condition 5a ensures that the new location, which T-Alloc adds to the focus, is not already contained in the panorama memory, which would violate condition 5b. To guarantee that condition 1c (alias-disjointness of focus memory types) holds after an **`alloc`**, we require the inequalities between the new variable and the focus to be represented in the alias information of the following command.

`load` needs the oracle conditions 6b and 6c to maintain condition 3 (E is typed according to Γ). Furthermore, in case a pointer was loaded, condition 6e is required to re-establish condition 6c.

Theorem 1 can trivially be extended to $\rightarrow^*$. It also follows that if a program of type τ starting in a consistent state terminates with value v, then we have $\vdash v : \tau$.

5.1.1 Properties of Type-Consistent States

Conditions 1 and 5 of type consistency give raise to some interesting properties which have already been mentioned in the introduction to the type system (Section 4.1).

An important fact is that if an alias token κ can be derived from an annotation G, and an environment E satisfies G, then E also satisfies κ. As a result, the sets of values obtained by looking up alias-disjoint sets of variables are disjoint if the lookup is performed in a satisfying environment.

Lemma 1 (Soundness of alias token derivation)**.** *Let $E \vDash G$ and $G \rhd \kappa$. Then $E \vDash \kappa$. Furthermore, let $G \rhd A_1 \boxdot A_2$. Then $E\ A_1 \cap E\ A_2 = \emptyset$.*

Now we can prove two important properties of type-consistent states: Two different memories are disjoint (Lemma 2), and a variable being accessible in a memory guarantees that it cannot be found in another memory (Lemma 3).

Lemma 2 (Different well-typed memories are disjoint)**.** *Assume a state consistent with a static context using oracle σ, i.e., we have $(E, L, S, U \mid s) : \tau \vDash_\sigma \Gamma, \Lambda, \Sigma$. Let $E \vDash G_s$ and $m_1, m_2 \in \operatorname{dom} \Sigma$ with $m_1 \neq m_2$. Then $\operatorname{dom}(S\ m_1) \cap \operatorname{dom}(S\ m_2) = \emptyset$, i.e., the memories denoted by these variables are disjoint.*

Proof. Since there is only one panorama memory, there are two cases to distinguish:

- One of the memories is the panorama memory. Assume without loss of generality that $\Sigma\ m_1 = \top$ and $\Sigma\ m_2 = A_2$. According to conditions 5b and 5c of state consistency, we have $\operatorname{dom}(S\ m_1) \cap E\ F_\Sigma = \emptyset$ and $\operatorname{dom}(S\ m_2) \subseteq E\ A_2$. Using the definition of F_Σ (Definition 11 on page 25), we can derive that $A_2 \subseteq F_\Sigma$ and hence $E\ A_2 \subseteq E\ F_\Sigma$. Thus we have $\operatorname{dom}(S\ m_2) \subseteq E\ F_\Sigma$ and therefore $\operatorname{dom}(S\ m_1) \cap \operatorname{dom}(S\ m_2) = \emptyset$.
- Otherwise, both memories are focus memories. Let $\Sigma\ m_1 = A_1$ and $\Sigma\ m_2 = A_2$. Using condition 5c we obtain $\operatorname{dom}(S\ m_1) \subseteq E\ A_1$ and similar for m_2 and A_2. From condition 1c we can derive $G_s \rhd A_1 \boxdot A_2$. Since E satisfies G, we have $E\ A_1 \cap E\ A_2 = \emptyset$ and hence $\operatorname{dom}(S\ m_1) \cap \operatorname{dom}(S\ m_2) = \emptyset$. □

Lemma 3 (Soundness of accessibility)**.** *Let $(E, L, S, U \mid s) : \tau \vDash_\sigma \Gamma, \Lambda, \Sigma$ and let $E \vDash G_s$. Let $m, m' \in \operatorname{dom} \Sigma$ with $m \neq m'$. Assume $G_s, F_\Sigma \rhd a \mathrel{\tilde{\in}} \Sigma\ m$. Then $E\ a \notin \operatorname{dom}(S\ m')$.*

Proof. The proof is done by case distinction over the type of m.

- Assume $\Sigma\ m = \top$. Then accessibility in m means $\forall a' \in F_\Sigma : G_s \rhd a \ncong a'$. Furthermore, m' is a focus memory and we have $\Sigma\ m' = A'$. Using condition 5c of the state consistency, we obtain $\operatorname{dom}(S\ m') \subseteq E\ A'$. Assume for contradiction that $E\ a \in \operatorname{dom}(\Sigma\ m')$. Then we obtain $E\ a \in E\ A'$ and hence there exists an $a' \in A'$ such that $E\ a = E\ a'$. However, we also have $A' \subseteq F_\Sigma$ and therefore $G_s \rhd a \ncong a'$. As shown in Lemma 1, using the fact that E satisfies G_s this implies $E\ a \neq E\ a'$, which is a contradiction.
- Assume $\Sigma\ m = A \neq \top$. Then accessibility means there exists an $a' \in A$ such that $G_s \rhd a \cong a'$. Using Lemma 1 and that E satisfies G_s, we obtain $E\ a = E\ a'$. We do a case distinction over the type of m'.
  - Let $\Sigma\ m' = \top$, so by condition 5b we obtain $\operatorname{dom}(S\ m') \cap E\ F_\Sigma = \emptyset$. Using $A \subseteq F_\Sigma$, we have $E\ a' \in E\ F_\Sigma$. Hence $E\ a = E\ a' \notin \operatorname{dom}(S\ m')$.
  - Otherwise, we have $\Sigma\ m' = A'$ and (by condition 1c) $G_s \rhd A \boxdot A'$, which implies $E\ A \cap E\ A' = \emptyset$ as E satisfies G_s. Assume for contradiction that

$E\ a \in \mathrm{dom}(\Sigma\ m')$. Then $E\ a \in E\ A'$ by condition 5c, but we also have $E\ a' \in E\ A$. Finally, we obtain $E\ a \neq E\ a'$ as both are contained in disjoint sets. This is in contradiction to the equality derived above. □

5.2 Semantic Equivalence and Normalisation

In this section, we give a sketch for the proof of the core property of the type system: If s is well-typed and alias-correct, then $norm_m\ s$ is well-typed as well, and both are semantically equivalent. You can find the definition of *norm* in Figure 3.5 on page 19.

In the following, we will often make use of the fact that normalisation does not change the alias annotation:

Lemma 4 (Normalisation preserves alias annotation)**.** *Let s be well-typed, i.e., assume $\Gamma, \Lambda, \Sigma \vdash s : \tau$. Then $G_{norm_m\ s} = G_s$.*

Proof. By induction over the structure of s. For most cases, $norm_m\ s$ immediately uses the annotation from s, so the statement is trivial. For `store` and `load`, we obtain $G = G_s$ from the fact that s is well-typed. This is the only case where we use the induction hypothesis. □

Preservation of well-typedness is straight forward to formalise:

Lemma 5 (Normalisation preserves well-typedness)**.** *Let $\Gamma, \Lambda, \Sigma \vdash s : \tau$. Then $norm_m\ s$ is well-typed, i.e., $\Gamma, \Lambda, \emptyset^m_\top \vdash norm_m\ s : \tau$.*

The proof is done by induction over the type derivation and the structure of s. Accessibility is easily proven in the normalised program since it only accesses the panorama memory, and the focus is always empty. The only interesting case is `alloc`. After the `merge`, only a panorama memory m is left, so the induction hypothesis for s can be used. The `merge` itself is also well-typed since it uses $G_s = G_{norm_m\ s}$ as annotation.

Proving the second part requires some preliminary definitions and lemmas. First of all, of course, we need a definition of semantic equivalence.

Definition 16 (Semantic equivalence)**.** Given a static context $(\Gamma, \Lambda, \Sigma)$ and return type τ, two programs s_1 and s_2 are *semantically equivalent* $(\Gamma, \Lambda, \Sigma \Vdash s_1 \approx s_2 : \tau)$ if either both programs produce the same result when started with environments both are consistent with, or both programs do not terminate:

$$\forall \sigma, Q :\ (Q \mid s_1) : \tau \vDash_\sigma \Gamma, \Lambda, \Sigma\ \wedge\ (Q \mid s_2) : \tau \vDash_\sigma \Gamma, \Lambda, \Sigma$$
$$\Rightarrow \forall v : (Q \mid s_1 \Downarrow v) \iff (Q \mid s_2 \Downarrow v)$$

and both programs are consistent with the same environments:

$$\forall \sigma, Q :\ (Q \mid s_1) : \tau \vDash_\sigma \Gamma, \Lambda, \Sigma \iff (Q \mid s_2) : \tau \vDash_\sigma \Gamma, \Lambda, \Sigma$$

We only consider states which are consistent with the given context as we make no attempt to guarantee anything for inconsistent executions. However, the first condition alone would result in a notion of semantic equivalence which is not transitive. Every ill-typed program would be equivalent to *every* program since there is no consistent execution for such programs. The second condition makes semantic equivalence a proper equivalence relation, as we proved in our Coq development.

The core theorem can now be formulated: An alias-correct program which is well-typed using no functions and only a single memory variable (the panorama memory m) is semantically equivalent to its normalisation.

Theorem 2 (Normalisation preserves semantics)**.** *Let* $\Gamma, \emptyset, \emptyset^m_\top \vdash s : \tau$ *and let* s *be alias-correct. Then* $\Gamma, \emptyset, \emptyset^m_\top \Vdash s \approx norm_m\ s : \tau$*.*

Proof of the second condition of $\approx$*.* The two states differ only in the program s. Using the assumption and Lemma 5, both s and $norm_m\ s$ are well-typed under $(\Gamma, \emptyset, \emptyset^m_\top)$. Since we have $G_s = G_{norm_m\ s}$ by Lemma 4, all the other conditions for consistency can be re-used, so the equivalence of consistency is easily shown. □

As a simple corollary, it follows from transitivity of semantic equivalence that two well-typed and alias-correct programs having the same normalisation are semantically equivalent.

If you compare above definition of semantic equivalence with the first definition of realisability (Definition 7 on page 18), you may notice that semantic equivalence of s and $norm_m\ s$ does *not* imply that s is realisable. This is because our first definition does not take the type system into account. Hence we update the definition to account for state consistency:

Definition 17 (Realisability, final)**.** Given a value type environment Γ and return type τ, a program s is *realisable* if the following holds for some m: Whenever an execution of s starting in a consistent state terminates with value v, then so does $norm_m\ s$ with the same initial environments.

$$\begin{aligned} \forall \sigma, Q :\ & (Q \mid s) : \tau \vDash_\sigma \Gamma, \emptyset, \emptyset^m_\top \\ & \Rightarrow \forall v : (Q \mid s \Downarrow v) \Rightarrow (Q \mid norm_m\ s \Downarrow v) \end{aligned}$$

Furthermore, there has to be at least one consistent state:

$$\exists \sigma, Q :\ (Q \mid s) : \tau \vDash_\sigma \Gamma, \emptyset, \emptyset^m_\top$$

We restrict the definition to only provide guarantees about executions consistent with the type environment, similar to how semantic equivalence does not take into account inconsistent executions. However, using just that condition would result in programs which do not even have a consistent state to be realisable, as there is no execution to reason about. Hence we require at least one consistent execution to exist.

Now it follows as a consequence of Theorem 2 that well-typed programs (under the given static context) are realisable. This relies on the fact that given a program which is well-typed in a static context, it is possible to synthesise a state consistent with this context.

Corollary 1 (Well-typed alias-correct programs are realisable)**.** *Let* $\Gamma, \emptyset, \emptyset^m_\top \vdash s : \tau$ *and let* s *be alias-correct. Then* s *is realisable.*

5.2.1 Proof of Semantic Equivalence to Normalisation

To prove the semantic equivalence of a program and its normalisation, we establish a relation between the states in the original execution (of the unmodified program) and the normalised execution. However, we cannot do a lock-step simulation proof here, i.e., a proof showing that if one program makes a step, then the other program makes a corresponding step [Ler09a, Sect. 3.7]. Since *norm* removes **`merge`** and **`split`**, it can be the case that the normalised program performs no step where the original program proceeded by one step. If the original program performed **`alloc`**, the normalised program needs to advance by two steps (**`alloc`** and **`merge`**) to catch up with the original execution.

Definition 18 (Normalised state)**.** Given a static context $(\Gamma, \Lambda, \Sigma)$, a state R_n is the *normalised state* of state R for memory variable m, written $\Gamma, \Lambda, \Sigma \Vdash R_n \sim_m R$, if all of the following hold:

1. $R_n.E = R.E$ (the value environments are equal)
2. For every $f \in \mathrm{dom}\ \Lambda$, let $c_n = R_n.L\ f$ and $c = R.L\ f$
   (a) $c_n.E = c.E \wedge c_n.\overline{x} = c.\overline{x}$ (closed variables and argument names are the same)
   (b) $c_n.m = m$ (the normalised function uses m as argument name for the memory)
   (c) $c_n.s = norm_m\ c.s$ (the code in the normalised closure is normalised)
3. $R_n.S\ m = \bigsqcup_{m' \in \mathrm{dom}\,\Sigma} R.S\ m'$ (m in the normalised state denotes the memory obtained by merging all original memories)
4. $R_n.U = R.U$ (the sets of used locations are the same)
5. $R_n.s = norm_m\ R.s$ (the normalised state contains the normalised program)

To show that this relation is maintained throughout the executions of both the original and the normalised program, we will use the following lemma, showing an important consequence of the two properties of type-consistent state proven in Section 5.1.1: If we have a proof of accessibility of a for a memory variable m in a state of the original program, then the memory $S\,m$ agrees with the merged memory μ at location $E\,a$.

Lemma 6. *Let* $(E, L, S, U \mid s) : \tau \vDash_\sigma \Gamma, \Lambda, \Sigma$ *and let* $E \vDash G_s$*. Let* $\mu = \bigsqcup_{m \in \mathrm{dom}\,\Sigma} S\,m$ *be the memory obtained by merging all available memories. Assume* $G_s, F_\Sigma \rhd a\ \tilde{\in}\ \Sigma\,m$*. Then* $\mu(E\,a) = (S\,m)(E\,a)$*, i.e.,* μ *and the memory denoted by* m *have the same value at the location described by* a*.*

Proof. We distinguish two cases:

- Assume $(E\,a) \notin \mathrm{dom}\,\mu$. Then $E\,a$ is not a valid location for any memory, and hence trivially $(S\,m)(E\,a) = \bot$.

- Otherwise, $(E\ a) \in \operatorname{dom} \mu$, so there must be some memory $m' \in \operatorname{dom} \Sigma$ with $(S\ m')(E\ a) = \mu(E\ a)$. Since memories assigned to different memory variables are disjoint by Lemma 2, there is exactly one such m'. If $m = m'$, we are trivially done. Otherwise, we can use Lemma 3 to show $E\,a \notin \operatorname{dom}(S\,m')$, which is a contradiction to $S\,m'$ and μ having the same value at $E\,a$. □

Now we can prove the induction step of the '⇒' part of semantic equivalence: If the original program takes a step, then the normalised program makes a corresponding series of steps. Since the notion of the corresponding normalised state relies on parts of the static context, we also need to include the new static context in the statement.

Lemma 7. *Let $R : \tau \vDash_\sigma \Gamma, \Lambda, \Sigma$ and let $R.E \vDash G_{R.s}$. Furthermore, assume a state R_n such that $\Gamma, \Lambda, \Sigma \Vdash R_n \sim_m R$ and let $R \to R'$. Then there exist a state R'_n, a memory oracle σ' and a static context $(\Gamma', \Lambda', \Sigma')$ such that $R' : \tau \vDash_{\sigma'} \Gamma', \Lambda', \Sigma'$ and $R_n \to^* R'_n$ and $\Gamma', \Lambda', \Sigma' \Vdash R'_n \sim_m R'$.*

The statement of this lemma is visualised in Figure 5.1 below, similar to the simulation diagrams by Leroy [Ler09a, Fig. 4]. Note that in our diagram, program execution progresses to the right, while Leroy chose programs to advance towards the bottom. Solid arrows represent execution steps, while dashed lines show which states correspond to each other. The `split` performed by the original execution has no corresponding step in the normalised execution, hence both original states correspond to the same normalised state. In case of `store`, both programs perform just one step. To catch up with the `alloc`, the normalised execution performs an additional `merge` before a correspondent state is reached. The `load` is again a single step for both programs.

The lemma can be proven by case distinction over the step performed in the original program. For `alloc`, the normalised program performs two steps so that the new location is merged into m. In case of `split` and `merge`, no normalised step is performed since both do not change the merged global memory μ: We can use $R'_n = R_n$. For `free`, `load` and `store`, Lemma 6 is used to justify the step of the normalised program. Using the preservation theorem, we know that type consistency is preserved in the original execution.

Based on this, we can prove the '⇒' part by doing induction over the execution of the original program. We know the original program to be alias-correct, so we have $R.E \vDash G_{R.s}$ throughout the execution. The step lemma above already provides us with the necessary proof of type consistency for the induction hypothesis. When the original execution reaches the return statement, we know the normalised execution is also at the

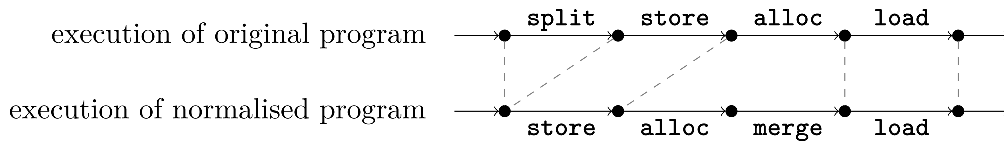


Figure 5.1: Visualisation of Lemma 7

same statement (*norm* does not change return statements) and the value environments agree, so the same value is returned.

Using a very similar induction, we can also show that if the normalised program is alias-correct, then so is the original one:

Corollary 2. *Let $norm_m\ s$ be alias-correct under static context $(\Gamma, \emptyset, \emptyset^m_\top)$ with type τ. Then s is alias-correct as well.*

To prove this, we perform induction on the execution of the original program which we are given. We have to show that on each state reached by this execution, the values in E satisfy the current alias annotation. We can give a corresponding execution of the normalised program, which we know to be alias-correct. Both end in states having the same value environments and annotations (remember that we have $G_s = G_{norm_m\ s}$ by Lemma 4). Since the tokens in the normalised program are satisfied, so are the original ones.

The '$\Leftarrow$' part of semantic equivalence is more complicated: When the normalised program performs a step, it could be in a state not corresponding to any original state if it just performed the `alloc`, but did not yet execute the succeeding `merge`. To compensate for this problem, we have to allow the normalised program to optionally perform a second step before we can give a corresponding original state. If such a step is performed, we do not allow the value environment E or the alias annotation to change.

Definition 19 (Optional step)**.** A state R'_n is an *optional step* away from state R_n, in symbols $R_n \rightarrow^? R'_n$, if

1. $R_n = R'_n$ (both states are the same) or
2. $R_n \rightarrow R'_n \wedge R_n.E = R'_n.E \wedge G_{R_n.s} = G_{R'_n.s}$ (a step was performed, but value environment and alias annotation are preserved)

The lemma used in the induction step of the '$\Leftarrow$' part can now be written down: If the normalised program makes a step, then after another optional step it reaches a state such that the original program can make a series of steps to the corresponding state.

Lemma 8. *Let $R : \tau \vDash_\sigma \Gamma, \Lambda, \Sigma$ and let $R.E \vDash G_{R.s}$. Furthermore, assume a state R_n such that $\Gamma, \Lambda, \Sigma \Vdash R_n \sim_m R$ and let $R_n \rightarrow R'_n$. Then there are states R''_n and R', a memory oracle σ' and a static context $(\Gamma', \Lambda', \Sigma')$ such that $R'_n \rightarrow^? R''_n$ and $R \rightarrow^* R'$ as well as $R' : \tau \vDash_{\sigma'} \Gamma', \Lambda', \Sigma'$ and $\Gamma', \Lambda', \Sigma' \Vdash R''_n \sim_m R'$.*

This lemma is visualised in Figure 5.2 on the next page. The original program has to perform the `split` before it can execute the `store` to reach a state corresponding to the normalise execution. In case of `alloc`, the normalised program has to perform another step to reach a state which has a counterpart in the original execution. `alloc` is the only case where $R'_n \neq R''_n$.

Again, the lemma is proven by case distinction over the step performed by the normalised program. Lemma 6 shows that if the memory access was valid in the normalised program (for the global memory), then it is also valid for the original program

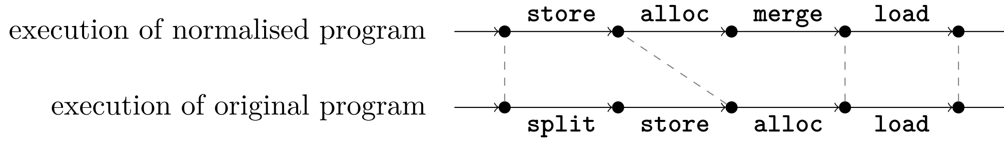


Figure 5.2: Visualisation of Lemma 8

(in a possibly restricted memory). For most cases, the optional step will not be taken and the original program performs just a single step. In case of **merge** or **split**, we have to perform steps in the original program until we reach a statement which is neither **merge** nor **split** (this requires an induction over the structure of the original program), then we reach the case of another statement. For **alloc**, the optional step is taken to compensate for the **merge** the normalised program performs: **merge** does not change the value environment E, and since the program is well-typed, it has the same annotation as the following statement. The original program performs a single step, then both are in sync again.

Using this, we can prove the '$\Leftarrow$' part of semantic equivalence by doing induction over the execution of the normalised program. The original program is alias-correct, so we know the value environments will satisfy the annotations, and the step lemma provides us with the type consistency proof for the induction hypothesis. If the optional step was taken, we make use of the fact that the semantics are deterministic, i.e., two states succeeding the same state are necessarily equal. Hence we can directly use the induction hypothesis. When the normalised execution reaches the return statement, we know the optional step was not taken since no step can be taken from there. Hence the original execution reached a state corresponding to the current normalised state, which implies that the value environments E are the same. After skipping **store** and **merge** statements in the original program, we know it must consist of just the return statement. Hence the same value is returned. This completes the proof of Theorem 2.

We can also use Lemma 8 to prove that if the original program is alias-correct, then so is the normalised one.

Corollary 3. *Let s be alias-correct under static context $(\Gamma, \emptyset, \emptyset^m_\top)$ with type τ. Then $norm_m$ s is alias-correct as well.*

Again we perform induction over the normalised execution, obtaining a corresponding original execution which we know to reach only states where the value environments satisfy the alias annotation. Hence the annotation is also satisfied by the current normalised state. If the optional step was taken, the same holds for the previous normalised state since both agree in terms of value environments and alias tokens.

Chapter 6

Evaluation

To evaluate the validity of our approach, we formalised IL/M and its properties in Coq. Furthermore, we manually performed an alias analysis on a simple program to ensure that real-world alias analysis is capable of providing the information we need.

6.1 Formalisation in Coq

The Coq formalisation of IL/M[1] is based on the work on IL/F by Schneider [Sch13]. Finite sets and maps (which are used to model environments and memories) are described by axioms. We also used axioms to describe expressions, values, types as well as evaluation and well-typedness of expressions, so that any concrete implementation satisfying these axioms can easily be used with our formalisation. Variable names are represented by natural numbers. In the Coq development, we proved all the lemmas, theorems and corollaries given in this thesis.

The formalisation is very close to the mathematical description presented here. The most important difference is the use of De Bruijn indices [Bru72] instead of names for closures. Function application uses a number to describe how many function declarations one has to 'go up' to find the one this application is referring to. Applying function 0 means performing direct recursion; function 1 is the most recently declared function etc. Instead of function (type) contexts, we use simple lists of closures and function types in states and static contexts, respectively.

Furthermore, the static context explicitly contains not only variable, closure and memory types but also the focus. While it would be possible to compute the focus from the memory types as we did in this thesis, this is very cumbersome in Coq. Instead, the focus is updated as needed by the inference rules for well-typedness, and type consistency requires the focus to be equal to the union of focus memory types.

The environments in the static context (for variable and memory types) use maps to `option` where `None` corresponds to $\bot$. For the dynamic environments, we map variable names directly to values and memories. We never actually need to know the domain

[1] You can find the source code on the CD containing the digital version of this thesis, or download it from `http://www.ralfj.de/cs/bachelor.zip`.

of these environments, so it is not a problem that all the 'other' variable names have associated values. Well-typed programs cannot access these names.

It turned out that goals and assumptions containing terms like `S[m <- mv][m']` are very common. Here, `S[m <- mv]` is our Coq notation for S_{μ}^{m}, i.e., updating a map. The notation `S[m']` performs a lookup in map `S`. We wrote a tactic called `destruct_lookup` which detects such terms and automatically distinguishes the cases `m = m'` and `m <> m'`. This is possible as maps require the domain type (in this case, variable names) to have a decidable equality. As a result, the term is simplified to `mv` and `S[m']`, respectively. Based on this, the tactic `crush` (inspired by Chlipala [Chl12]) repeatedly tries to apply `destruct_lookup` and a few other automation tactics provided by Coq like `eauto`, `discriminate`, `autoinjections` and `subst` until it solves the goal and all new sub-goals. This tactic was very useful since it is even able to solve goals containing several updates (e.g. `ST[m1 <- None][m2 <- Some MemPanorama][m']`) making use of previously established knowledge like `ST[m1 <- None][m'] = None`.

Based on this, defining the inference rules for the semantics (file `ILM.v`) and the type system (file `ILM_Types.v`) as inductive types in Coq is straight-forward. The proof of type preservation (Theorem 1) can be found under the name `preservation` (file `ILM_Preservation.v`). This proof is very long and makes heavy use of `crush`, but it contains only few surprises. The most complicated part was proving that the oracle still applies (condition 6 of Definition 14) after performing `alloc`, which requires reasoning about why old pointers cannot point to the new location.

Proving that well-typed programs are semantically equivalent to their normalisation (Theorem 2) was more sophisticated due to the complicated inductions. The Coq proof is called `normalise_equiv` (file `ILM_Equivalences.v`). Getting the induction hypotheses right turned out to be quite complicated, so that invoking the actual `induction` tactic is preceded by duplicating assumptions and variables, performing the right substitutions. Then we carefully `clear` assumptions to obtain a usable induction hypothesis.

Coq does not deal well with tuples: Often we would give a name to the entire tuple, so that quantifying over two states does not involve ten variable names. Then, however, these tuples have to be destructed before the components can be properly used. Since Coq does not choose useful variable names, this means our code contains the same long destruct patterns over and over again. Lemmas and definitions formulated using the components require destructing the tuple before they properly simplify, so this destruction could often not be avoided.

On the positive side, we made heavy use of Coq existential variables, especially of `eapply`. In many cases, we could avoid referencing assumptions by name since Coq could figure out automatically which one to use. This made it much easier to use the same or very similar tactics for many different cases, especially when doing case distinction over the structure of a program, a step, or a typing derivation, which leads to more than ten cases.

For these large case distinctions, the new proof structure features of Coq 8.4 came in very handy. They make such long proofs much more readable and easier to maintain as definitions change.

Furthermore, we proved a few simple semantic equivalences which we justified in this thesis using the graph representation of a program: Two **store** operations performed on different memory variables can be swapped without changing the program behaviour (file ILM_Equivalences.v). The same holds when swapping **store** and **load**, **store** and **free** or **free** and **load**.

The Coq files come with comments explaining which definitions and lemmas in this thesis the Coq declarations correspond to.

6.2 Reality Check

Throughout this thesis, we assumed that we have an alias analysis available which is capable of enriching the program with equalities and inequalities of pointer values that hold for each execution. Such information is crucial to be able to perform any re-ordering of memory operations. We performed a manual analysis on a simple program to ensure real-world alias analyses can provide enough information. Robert and Leroy [RL12] describe an alias analysis on RTL, one of the intermediate languages used by the CompCert C compiler [Ler09a]. RTL is a low-level imperative language without explicit memory. It operates on pseudo-registers of which each function has an unlimited supply.

The analysis is proven sound in Coq, and it is able to deal with dynamically allocated memory. It computes, for each program point, a set of abstract locations a register can point to. It also maintains an abstract memory state, mapping abstract pointers to sets of abstract locations, to have information available for pointers obtained via **load**. However, this is not needed for our example, so we omit it in the following. Abstract locations are identified by an abstract block and an offset into that block, which can be unknown. Blocks include the current Stack frame, Allocs(All) to denote the entire memory area allocated by the current execution of the function, and Allocs(Just l_i) to describe the memory area allocated at the given line during the current execution of the function (which can be arbitrary many locations, if that line is within a loop).

We used the following simple program, an extended imperative implicit-memory version of the example used throughout Section 4.1:

```
a = alloc();
b = alloc();
store(a, v);
store(b, w);
x = load(c);
```

Note that above syntax is not actual RTL: It is straight forward to translate to RTL, and we do not care about the low-level details necessary to access memory in RTL (like the word size, i.e., the amount of bits read from memory).

The information obtained by the alias analysis about this program is given in Table 6.1 on the following page. The offset is always 0 since IL/M does not support compound data types, so the allocated memory block is a singleton block (with just one location).

abstract register state before statement	
	`a = alloc();`
$\mathtt{a} \leftarrow \{(\mathtt{Allocs}(\mathtt{Just}\ l_1), 0)\}$	`b = alloc();`
$\mathtt{a} \leftarrow \{(\mathtt{Allocs}(\mathtt{Just}\ l_1), 0)\}, \mathtt{b} \leftarrow \{(\mathtt{Allocs}(\mathtt{Just}\ l_2), 0)\}$	`store(a, v);`
$\mathtt{a} \leftarrow \{(\mathtt{Allocs}(\mathtt{Just}\ l_1), 0)\}, \mathtt{b} \leftarrow \{(\mathtt{Allocs}(\mathtt{Just}\ l_2), 0)\}$	`store(b, w);`
$\mathtt{a} \leftarrow \{(\mathtt{Allocs}(\mathtt{Just}\ l_1), 0)\}, \mathtt{b} \leftarrow \{(\mathtt{Allocs}(\mathtt{Just}\ l_2), 0)\}$	`x = load(c);`

Table 6.1: Results of Robert and Leroy's alias analysis on the sample program

The alias analysis comes with a notion of *disjoint abstract locations*, where two abstract locations are disjoint if they refer to disjoint abstract blocks, or if the offsets are both known and different from each other. Two sets of abstract locations are disjoint if all the locations they contain are pairwise disjoint. Since abstract blocks of memory allocated at different program points are disjoint, the sets of abstract locations for `a` and `b` are disjoint.

Using the corollary `nonaliasing_sound` of the soundness of the alias analysis [RL12, Sect. 5], the concrete values of registers with disjoint sets of abstract locations are different for each execution. Hence adding the token $\mathtt{a} \not\cong \mathtt{b}$ to the corresponding IL/M program is justified, which allows making the **`store`** operations independent of each other (see Section 4.1).

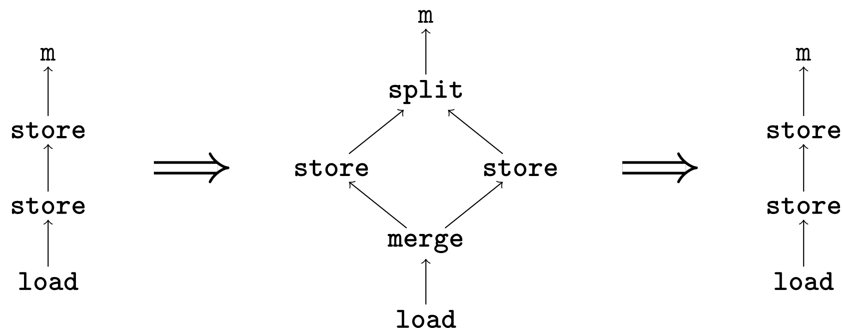
Chapter 7

Conclusion

We described a language called IL/M which is capable of representing independence of memory operations. Independent operations can easily be reordered, no further analysis is required to ensure that the behaviour of the program is preserved. The language uses alias information (equalities and inequalities of pointer values holding for each execution) to justify this independence. Based on that information, we presented a type system for IL/M which guarantees that well-typed programs are easy to translate to machine code despite the functional approach we take to model memory.

Furthermore, the type system provides support for program transformations which mark memory operations independent. The program graph induces a partial order such that independent operations are not ordered. The typing relation can provide semantic guarantees by restricting which variables can be used to **load** or **store** in a memory: A memory access is allowed only if the annotated alias information provides a proof that unordered accesses do not interfere, i.e., they do not affect the same memory location.

If a transformation relaxes the order of memory operations, it can prove the resulting program to be semantically equivalent to the original one by proving its well-typedness. This is achieved by defining a general transformation called *normalisation* which makes each memory operation depend on the previous one, and proving that a well-typed program and its normalisation exhibit the same behaviour. Relaxing dependencies does not alter the normalisation, hence the desired semantic equivalence follows immediately.



Another consequence is that well-typed programs can be easily executed on real machines. Transforming the program such that memory operations are linearised does not change the normalisation, and translating the product of this transformation to machine code with instructions to access the global memory is straightforward.

7.1 Limitations and Future Work

Currently, IL/M forbids pointer arithmetic to keep the memory model simpler. In order to actually model C, the language needs to be extended to support calculations on pointers. As mentioned in Section 5.1, this requires the types of values stored in memory cells to be tracked in the semantics, since the result of a **`load`** depends on the type used to access memory. Using a type incompatible with the value stored at the given location results in the program getting stuck. This approach is used by the CompCert verified C compiler [LB08]. The memory oracle we introduced to formulate type consistency would no longer be needed. However, we do not expect this to affect the alias annotations and the requirements the type system imposes to provide its guarantees with regard to normalisation.

Extending functions to support arbitrary memory arguments would allow the program structure to express that a loop (which in IL/M is translated to recursive function application) affects only a small part of the memory. The remaining memory can be forwarded unaltered. Currently, IL/M enforces merging all memory before calling a function, i.e., before entering the loop body. This extension severely complicates handling function application in the type system, but we expect the remainder of the type system and the consistency relation to need few changes.

Furthermore, IL/M does not provide for compounds, i.e., structures and arrays. However, since we did not make explicit which values the language works on, compounds could be handled completely hidden from IL/M. Based on the extension for pointer arithmetic, expressions could be used to obtain values, including pointers, from a compound value.

It may also seem that alias annotations have to be extended to be able to refer to values stored inside a compound. However, we believe this is not the case: The language requires a program to first obtain the value from a compound before doing memory access with it. Expressions are not allowed as arguments for **`store`** or **`load`**. Hence the task of tracking values of pointers stored in compound values or in the memory can be performed by the alias analysis. It can annotate the equalities and inequalities it inferred once the pointer is stored in a simple variable. The proof of correctness of that annotation then still has to deal with where that pointer value comes from, but for the type system it is only relevant that the required (in)equalities hold once the memory access is performed.

An alternative approach to introduce compounds, which allows pointers addressing parts of a compound (like elements of a structure), would be to make them explicit in IL/M. In this case, types would need more structure to represent these constructions. Furthermore, the syntax would be extended by a new statement to obtain the address of an element, given the address of the compound value. Concerning the alias annotations, the same considerations as above apply.

Bibliography

[All70] Frances E. Allen. 'Control flow analysis'. In: *SIGPLAN Not.* 5.7 (July 1970), pp. 1–19.

[ASU86] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[BBZ11] Matthias Braun, Sebastian Buchwald and Andreas Zwinkau. *Firm — A Graph-Based Intermediate Representation*. Tech. rep. 35. Karlsruhe Institute of Technology, 2011.

[Bru72] N. G. De Bruijn. 'Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem'. In: *Indagationes Mathematicae* 34 (1972), pp. 381–392.

[C11] JTC1/SC22/WG14 – ISO C Working Group. *ISO/IEC 9899:2011*. 8th Dec. 2011.

[Chl12] Adam Chlipala. *Certified Programming with Dependent Types*. URL: `http://adam.chlipala.net/cpdt/`.

[Cou86] Deborah S. Coutant. 'Retargetable high-level alias analysis'. In: *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. POPL '86. New York, NY, USA: ACM, 1986, pp. 110–118.

[CP08] Arthur Charguéraud and François Pottier. 'Functional translation of a calculus of capabilities'. In: *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*. ICFP '08. New York, NY, USA: ACM, 2008, pp. 213–224.

[CP95] Cliff Click and Michael Paleczny. 'A simple graph-based intermediate representation'. In: *Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*. IR '95. New York, NY, USA: ACM, 1995, pp. 35–49.

[Gir87] J.-Y. Girard. 'Linear Logic'. In: *Theoretical Computer Science* 50.1 (1987), pp. 1–102.

[Hoa69] C. A. R. Hoare. 'An axiomatic basis for computer programming'. In: *Communications of the ACM* 12.10 (1969), pp. 576–580.

[LB08] Xavier Leroy and Sandrine Blazy. 'Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations'. In: *Journal of Automated Reasoning* 41.1 (July 2008), pp. 1–31.

[Ler09a] Xavier Leroy. 'A Formally Verified Compiler Back-end'. In: *Journal of Automated Reasoning* 43.4 (Dec. 2009), pp. 363–446.

[Ler09b] Xavier Leroy. 'Formal verification of a realistic compiler'. In: *Communications of the ACM* 52.7 (July 2009), pp. 107–115.

[Mal08] Christoph H. Mallon. 'Registerdruckgewahre Befehlsanordnung'. Diploma Thesis. Universität Karlsruhe, 2008.

[Muc97] Steven S. Muchnick. *Advanced compiler design and implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[OBM90] Karl J. Ottenstein, Robert A. Ballance and Arthur B. MacCabe. 'The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages'. In: *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*. PLDI '90. New York, NY, USA: ACM, June 1990, pp. 257–271.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. Cambridge, MA, USA: MIT Press, 2002.

[Pro59] Reese T. Prosser. 'Applications of Boolean matrices to the analysis of flow diagrams'. In: *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*. IRE-AIEE-ACM '59 (Eastern). New York, NY, USA: ACM, 1959, pp. 133–138.

[Ram94] G. Ramalingam. 'The Undecidability of Aliasing'. In: *ACM Transactions on Programming Languages and Systems* 16.5 (1994), pp. 1467–1471.

[Rey02] John Reynolds. 'Separation Logic: A Logic for Shared Mutable Data Structures'. In: IEEE Computer Society, 2002, pp. 55–74.

[RL12] Valentin Robert and Xavier Leroy. 'A formally-verified alias analysis'. In: *Proceedings of the Second international conference on Certified Programs and Proofs*. CPP'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 11–26.

[Sch13] Sigurd Schneider. 'Semantics of an Intermediate Language for Program Transformation'. In preparation. Master's Thesis. Universität des Saarlandes, 2013.

[Ste95] Bjarne Steensgaard. 'Sparse Functional Stores for Imperative Programs'. In: *ACM SIGPLAN Workshop on Intermediate Representations*. 1995, pp. 62–70.

[Str00] Christopher Strachey. 'Fundamental Concepts in Programming Languages'. In: *Higher-Order and Symbolic Computation* 13.1-2 (Apr. 2000), pp. 11–49.

[SWH09] H. Seidl, R. Wilhelm and S. Hack. *Übersetzerbau: Band 3: Analyse und Transformation*. eXamen.press. Berlin, Heidelberg: Springer-Verlag, 2009.

[Wad93] Philip Wadler. ‘A Taste of Linear Logic’. In: *Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science*. MFCS ’93. London, UK: Springer-Verlag, 1993, pp. 185–210.

[WCM00] David Walker, Karl Crary and Greg Morrisett. ‘Typed Memory Management in a Calculus of Capabilities’. In: ACM Press, 2000, pp. 262–275.