Deductive program verification for a language with a Rust-like typing discipline

Xavier Denis

Deductive program verification for a language with a Rust-like typing discipline

Xavier Denis [1]

[1] *Université Paris-Saclay, CNRS, Inria, Laboratoire de recherche en informatique, 91405, Orsay, France.*

October 8, 2020

Abstract

Deductive program verification seeks to eliminate bugs in software by translating programs annotated with specifications into logical formulas which are then solved using semi-automated tools. When verifying programs using a mutable heap, it is often required to show that pointers do not alias each other, ensuring there is only one way to modify structures in memory. This leads to cumbersome proof obligations and makes verification much more challenging. Newer languages like Rust feature pointers as well but prevent aliasing through the type system. This opens the door to simpler approaches to verification, free of tedious proof obligations.

We propose a technique for the verification of Rust programs by translation to a functional language. The challenge of this translation is the handling of mutable borrows, pointers with control of aliasing in a region of memory. To overcome this, we used a technique inspired by prophecy variables to predict the final values of borrows. The main contribution of this work is to prove this translation correct. We developed a proof-of-concept tool to show the viability of this approach.

1 Introduction

Over the past 50 years programming languages have made major strides, allowing programmers to reason about and abstract ever larger software projects. Yet, when performance and efficiency become concerns, they resort to low-level programming languages like C/C++ or even assembly. These languages offer control over memory and in particular allow unrestricted usage of *pointers* to build complex data structures and efficient algorithms. This power comes at a cost. Reasoning about the correctness of pointer programs is very tricky. The challenge is *aliasing*, when a value can be accessed through more than one name. When two variables are aliased, changing either also changes the other. This makes it difficult to reason about code because programmers must keep in mind all potential aliases to understand the state of their programs.

Languages like C which have pervasive aliasing make this incredibly challenging and it often is a source of bugs in software. For example, Figure 1, performs a swap using XOR to avoid allocating a temporary variable[3]. But there's a bug! If the user provides the same pointer to both arguments, this function will write 0 to it instead. Aliasing and related issues like buffer overflows occur constantly, and cause many safety and correctness issues in C software.

In most programs, programmers make the implicit assumption that values are not aliased, but in C the compiler offers no help to verify this. When attempting to *formally verify* a C program, such aliasing assumptions must become explicit, and it can easily turn into a nightmare [5].

```
void XorSwap( int* x, int* y ) { *x ^= *y;  *y ^= *x;  *x ^= *y; }
```

Figure 1: Swap implemented with XOR which assumes arguments are non-aliased

```
fn xor_swap(x: &mut i32, y: &mut i32) {
  *x ^= *y; *y ^= *x; *x ^= *y; }
```

Figure 2: Translation of program of Figure 1 to Rust

Many approaches exist in the literature to overcome aliasing issues in the formal verification of programs in C-like languages. This work takes another path: instead considering a language like Rust, which offers a strong type system to control aliasing.

1.1 The Rust Programming Language

The Rust programming language is a recent entry in the field of systems programming languages. Its 1.0 release, published in 2015[2] aims to "empower everyone to build reliable and efficient software"[1]. Rust provides memory-safety without requiring a garbage collector through its type system[11]. The type system uses a system of *ownership* to ensure that mutable references cannot alias. When the `xorSwap` program of Figure 1 is translated to Rust, as shown in Figure 2, the bug is no longer possible: a call to that function using the same argument twice will trigger a type checking error.

Ownership in Rust In Rust every memory cell has a unique *owner*, which has exclusive read and write permissions. This is sufficient to program in Rust using a functional programming style. The program of Figure 3 illustrates this: it constructs a pair, and then increments the second component, not by mutation but by producing a new pair. But this style is limiting and more imperative programming style is desirable.

Borrows and Lifetimes To perform memory mutation, Rust uses safe pointers called *references* that can *borrow* these permissions from their owner. But then aliasing arises and it is where the Rust-specific type system comes into play: such a borrow restricts the permission of the original owner for a limited period of time so that aliasing is controlled. When a reference is created it can come in one of two forms: either unique and mutable or shareable but immutable. The Rust compiler infers the *lifetime* of references after which the permissions are restored to the owner.

The program of Figure 4 illustrates this, exploiting the ownership and borrowing mechanisms of Rust. When `x` is mutably borrowed into `y`, it transfers its read and write permissions to `y` and so loses access to its own contents. The function `inc`, takes ownership of its argument, *consuming*

```
fn main () {
  let x = Box::new((1,2));
  let x = Box::new((x.0, x.1 + 1));
  assert_eq!(x.1, 3);
}
```

Figure 3: Updating a box functionally

```
fn main() {
    let mut x = 0;
    let y = &mut x;
    inc(y); inc(y);
    assert_eq!(*y, 2);
}
```

```
fn inc(x: &mut i32) {
    *x += 1;
}
```

Figure 4: Incrementing a reference twice

it, so Rust implicitly *reborrows* `y` before calling `inc`. Reborrowing consists in borrowing the permissions of a borrow for a shorter lifetime than the original borrow. This mechanism makes borrows flexible and intuitive to use, because after each call to `inc`, morally we should be allowed to use `y` again. During compilation Rust statically verifies that borrows are used correctly, inferring the lifetime of each borrow.

1.2 Verification

What does it mean for a program to be correct? A common answer is that the program should satisfy a *logical specification*. This specification encodes the properties expected of a program. For example, a specification of `xor_swap` could be that the values of `x` and `y` are exchanged. To verify that programs satisfy specifications many techniques can be used, from model checking[7], to dependent types[10] or deductive verification[9].

Deductive verification of programs works by translating a program and its specification into a collection of *verification conditions* such that their truth implies the program follows its specification. These verification conditions can be discharged using a variety of tools, though commonly, automated SMT solvers are used. This highly automated approach to program verification is appealing because it allows engineers to focus their attention on developing a specification rather than proving logical formulas.

Verification conditions can be generated by several techniques, including the *weakest precondition calculus*. This approach starts with the postcondition of a specification and asks what the weakest precondition that upholds the postcondition is. A rule is determined for each syntactic element of the language, and then recursively applied to translate a whole program to this logical form.

Deductive verification tools for languages with aliasing like C generate non-aliasing conditions[9] as part of the verification. By leveraging the ownership property of Rust programs, we can eliminate these conditions and simplify the verification conditions required to prove specifications true.

1.3 Contributions

We present a schema for the deductive verification of Rust programs by translation to a functional language. This translation handles both mutable and immutable borrows as well as owned pointers. This paper is structured in several sections, Section 2 presents μMIR, a fragment of Rust allowing only owned data. Using μMIR, we detail the type system which enforces ownership and its semantics. We then present a schema for the verification of μMIR and prove it correct.

The Section 3 section extends μMIR with mutable and immutable references into MiniMir. We explain how the introduction of references impacts the type system and semantics. Then we present a translation of MiniMir to a functional language with non-determinism and extend the proof of μMIR to handle references.

⟨*Instruction*⟩ ::= ⟨x⟩ '`:=`' '`box`' ⟨x⟩
| ⟨x⟩ '`:=`' '`unbox`' ⟨x⟩ | ⟨x⟩ '`:=`' '`copy`' ⟨x⟩
| ⟨x⟩ '`:=`' '`copy`' '`*`' ⟨x⟩
| '`drop`' ⟨x⟩ | '`swap`' '`(`' ⟨x⟩ '`,`' ⟨x⟩ '`)`'
| ⟨x⟩ '`:=`' ⟨x⟩ ⟨*op*⟩ ⟨x⟩ | ⟨x⟩ '`:=`' ⟨*const*⟩
| '`(`' ⟨x⟩ '`,`' ⟨x⟩ '`)`' '`:=`' ⟨x⟩ | ⟨x⟩ '`:=`' '`(`' ⟨x⟩ '`,`' ⟨x⟩ '`)`' | ⟨x⟩ '`:=`' '`inj`'$_i$ ⟨x⟩
| '`call`' f (⟨x⟩, .., ⟨x⟩)
| '`assert`' ⟨x⟩

⟨*Statement*⟩ ::= ⟨*Instruction*⟩ '`;`' '`goto`' ℓ
| '`match`' ⟨x⟩ '`{`' inj_0 ⟨x⟩ $\to$ '`goto`' ℓ, inj_1 ⟨x⟩ $\to$ '`goto`' ℓ '`}`'
| '`return`' ⟨x⟩

⟨*LabelledStatement*⟩ ::= ℓ '`:`' ⟨*Statement*⟩

⟨*const*⟩ ::= n | () | true | false

⟨*op*⟩ ::= '`+`' | '`-`' | '`<`' | '`>`' | '`=`'

⟨*Var, x*⟩ ::= $\text{s}^{\langle T \rangle}$

⟨*Type, T*⟩ ::= ⟨*Type*⟩ × ⟨*Type*⟩ | ⟨*Type*⟩ + ⟨*Type*⟩ | '`box`' ⟨*Type*⟩ | bool | unit | int

⟨*Signature*⟩ ::= '`fn`' name '`(`' s : ⟨T⟩ '`,`' ... '`,`' s : ⟨T⟩ '`)`' '`->`' ⟨T⟩

Figure 5: Grammar of μMIR

In Section 4 the translation, we developed a proof-of-concept implementation and showed it is capable of verifying real Rust programs.

2 A basic language with ownership

Rust is a complex programming language, targeted at industrial use with a large syntax and underspecified operational semantics. During compilation, the Rust compiler translates programs to MIR, an intermediate language with a greatly simplified syntax and a *graph* structure. Since 2018, with the introduction of *non-lexical lifetimes*, the rules for borrow checking are formulated on the MIR graph rather than source Rust. This makes MIR an attractive starting point for verification of Rust since, with its smaller syntax and simpler static analysis, translations and proofs stay smaller. We begin by formalizing a fragment of MIR containing only owned values called μMIR. This restriction makes it impossible to express many programs but gives room to focus on the other defining characteristic of μMIR, its graph structure.

2.1 The μMIR language

A program $\mathcal{P}$ is a collection of *function declarations*, and must contain a function `main` which acts as the entry point. Function declarations are in turn composed of a triple (Δ, ℓ, σ) consisting of a set of *labeled statements* Δ, an entry label ℓ, and a *function signature* σ. Function signatures are made of the function name, a collection of input parameter names and types and the return type. The syntax of μMIR is described in Figure 5.

Statements come in three forms: an instruction followed by a goto, a match on a variable or a return. We write $\mathcal{P}_{f,\ell}$ to refer to the statement associated with the label ℓ of the function f in program $\mathcal{P}$. Instructions perform the basic operations of the language. In each instruction the variables are annotated with their types. There are instructions to allocate, dereference and deallocate an owned pointer on the heap. In μMIR a `box` type is an owned pointer to a value, granting the owner exclusive access to the pointed region of memory. Values of certain types can be copied, and the contents of a box can be as well. Appendix A includes the rules for determining if a value can be copied. Dropping is the mechanism through which variables are freed. In μMIR, writing to memory is done by swapping values. Finally, primitive operations like assigning literals, arithmetic and comparisons are supported. μMIR also includes instructions to interact with products and sums.

```
ℓ0: t1 := 1; t2 := 2; t3 := (t1, t2); x := box t3; goto ℓ4
ℓ4: t4 := unbox x; (t5, t6) := t4; t7 := t6 + 1; goto ℓ5;
ℓ5: t8 := (t5, t7); x := box t8; goto ℓ9;
ℓ9: t9 := unbox x; (x1, x2) := t9; goto ℓ12
ℓ12: t10 := 3; t11 := x2 = t10; assert t11; goto ℓ14
ℓ14: drop x2; drop t10; drop t11; goto ℓ17
ℓ17: ret := (); return ret
```

Figure 6: Figure 3 translated to μMIR

Figure 6 displays how the program of Figure 3 is expressed in μMIR. To save space we put several statements on a single line. The code size blows up but decomposes the individual operations that happened in Figure 3 in a manner very similar to actual MIR.

2.2 A type system for ownership

The type system of μMIR is responsible for enforcing the unique ownership of every value in a program. Operations which manipulate boxes consume their inputs, removing them from the typing context and introducing new bindings for the results. Because of the imperative, graph structure of μMIR, the typing judgements take a different form from most languages. The presence of cycles within a graph make traditional natural deduction trees impossible to use. Instead, each program point is assigned a typing context, and typing relates the contexts before and after the evaluation of a statement.

Each label ℓ of a function is associated with a *partial variable context* $\mathbf{\Gamma}$. The partial variable context is a collection of items of the form $x : T$, where x is a variable name and T is a type. In a μMIR program $\mathcal{P}$, each function f is given a *whole context* $\mathbf{\Xi}$ which is the collection of the partial contexts for all labels in f. Additionally, we use $\mathbf{\Sigma}$ to refer to the collection of signatures found in a program.

Typing judgements for instructions have the form $\mathbf{\Gamma} \vdash_f I \dashv \mathbf{\Gamma}'$, which relate the partial contexts before and after executing the instruction I in the function f. In this representation, the mechanism of ownership becomes clearer, for example looking at (BOX) shows how when a `box` is created, the original variable y is destroyed to ensure the new pointer x remains the unique owner.

$$\frac{}{\mathbf{\Sigma}; y : T, \mathbf{\Gamma} \vdash_f x^{\texttt{box}\, T} \texttt{:= box } y^T \dashv \mathbf{\Gamma}, x : \texttt{box } T} \text{ (BOX)}$$

Typing judgements for statements have the form $\mathbf{\Sigma}; \mathbf{\Xi}; \mathcal{P} \vdash_{f,a} S$ and verify that the statement with label a in function f, types with the expected partial context.

$$\frac{\mathbf{\Sigma}; \mathbf{\Xi}_a \vdash_f I \dashv \mathbf{\Sigma}; \mathbf{\Xi}_\ell}{\mathbf{\Sigma}; \mathbf{\Xi}; \mathcal{P} \vdash_{f,a} I \texttt{; goto } \ell} \text{ (SEQUENCE)}$$

The rule for (BOX) checks that the instruction I can be typed using $\mathbf{\Xi}_a$, producing $\mathbf{\Xi}_\ell$. In this approach, each statement checks an edge of the μMIR graph. The rules for (FUNCTION) and (PROGRAM) check that each function forms a graph. The full typing rules can be found in Appendix A.

2.3 Operational Semantics of μMIR

The operational semantics of μMIR is given by a reduction relation over an abstract heap machine. The configurations of the μMIR machine are of the form $\langle f; \ell \mid \mathbf{S} \mid \mathbf{F} \mid \mathbf{H} \rangle$, where f is the name of the function being executed and ℓ is a label in f, $\mathbf{S}$ is the *call stack*, $\mathbf{F}$ is the *frame*, and $\mathbf{H}$ is the *heap*. The stack is composed of triples $[f; \ell, x, \mathbf{F}]$ of a return label, a variable name for the result and a frame. A frame $\mathbf{F}$ is a partial function from variable names to *addresses*. We support pointer arithmetic: adding an address and an integer to obtain an address. The heap maps those addresses to a value which is either: an address, an integer, a boolean value, or unit. Complex values such as pairs or sums are stored in contiguous regions of the heap. Operations which must move or copy these values calculate the *size* of the values from their types:

$$\begin{aligned} |T_1 \times T_2| &= |T_1| + |T_2| \\ |T_1 + T_2| &= 1 + max(|T_1|, |T_2|) \\ |\texttt{box}\ T| &= |bool| = |int| = |unit| = 1 \end{aligned}$$

Definition 2.1 (Notations)**.** *If F is a partial function, and A is a subset of its domain, then $A \triangleleft F$ denotes the* domain restriction *removing A from the domain of F, and is defined as*

$$A \triangleleft F = \{(x, v) \mid (x, v) \in F, x \notin A\}$$

Below, we include the rule for `drop`, which deallocates a variable. When a variable is dropped, we lookup it's address in the current frame and remove it from the heap. The remaining reductions are given in Appendix B.

$$\frac{\mathcal{P}_{f;\ell} = \texttt{drop}\ x^T\texttt{; goto}\ \ell\texttt{'}}{\langle f; \ell \mid \mathbf{S} \mid \mathbf{F} \oplus \{(x, a)\} \mid \mathbf{H} \rangle \rightarrow_{\mathcal{P}} \langle f; \ell' \mid \mathbf{S} \mid \mathbf{F} \mid \{a\} \triangleleft \mathbf{H} \rangle}\ (\textsc{drop})$$

2.4 Type preservation by reduction

We would like to ensure that the only way for a well-typed μMIR program to get stuck is by failing an assertion. This property allows us to reduce the safety of a program to the validity of its assertions. But if we look at the semantics from Section 2.3, it is clear that the machine can easily get stuck when the frame and heap contain invalid addresses or missing data. To correct this, we define a notion of *well-typed configurations* which restrict us to only the configurations that could appear when evaluating a well-typed program. We can then prove the standard preservation of typing for these configurations, showing that for well-typed programs only assertions can block evaluation.

Definition 2.2 (Heap Fragment Type)**.** *When we write $\mathbf{H} \vdash a : T$, we mean to say that the heap fragment $\mathbf{H}$ starting at location a, corresponds to a value of type T.*

$$\frac{\mathbf{H}_1 \vdash a : T_1 \qquad \mathbf{H}_2 \vdash a + |T_1| : T_2 \qquad \mathbf{H} = \mathbf{H}_1 \oplus \mathbf{H}_2}{\mathbf{H} \vdash a : T_1 \times T_2}$$

$$\frac{\mathbf{H} \vdash a + 1 : T_i}{\mathbf{H} \oplus \{(a, i)\} \vdash a : T_0 + T_1} \qquad \frac{\mathbf{H} \vdash p : T}{\mathbf{H} \oplus \{(a, p)\} \vdash a : \texttt{box}\ T}$$

$$\frac{c\ \textit{is a literal of}\ T \qquad T \in \{\texttt{int}, \texttt{bool}, \texttt{unit}\}}{\{(a, c)\} \vdash a : T}$$

A well-typed configuration is just a configuration where the heap $\mathbf{H}$ contains exactly the variables accessible in the frame $\mathbf{F}$.

Definition 2.3. *A frame* $\mathbf{F}$ *is well-typed at* $f; \ell$ *for a portion of a heap* $\mathbf{H}$ *if* f *is typed with* Ξ *in* $\mathcal{P}$*, and:*

$$\mathbf{H} = \bigoplus_{x \in \mathbf{F}} \mathbf{H}_x \qquad\qquad dom(\mathbf{F}) = dom(\Xi_\ell)$$

$$\forall x \in \mathbf{F}, \mathbf{H}_x \vdash x : \Xi_\ell(x)$$

A configuration $\langle f; \ell \mid \mathbf{S} \mid \mathbf{F} \mid \mathbf{H} \rangle$ *is well-typed for a well-typed program* $\mathcal{P}$ *when it satisfies the following conditions:*

$$\forall i \in [1, |\mathbf{S}|],\ \mathbf{S}[i] = [f_i; \ell_i, x_i, \mathbf{F}_i] \Rightarrow \textit{well-typed}\ \mathbf{F}_i\ f_i; \ell_i\ \mathbf{H}_i$$

$$\textit{well-typed}\ \mathbf{F}\ f; \ell\ \mathbf{H}_0$$

$$\mathbf{H} = \mathbf{H}_0 \oplus \bigoplus_{i \in [1, |\mathbf{S}|]} \mathbf{H}_i$$

We can now define what it means for reduction to preserve types, it simply ensures that the heap is composed exactly of the values expected at a given program point.

Theorem 2.4 (Type Preservation)**.** *Given a well-typed configuration* C*, if* $C \to_{\mathcal{P}} C'$*, then* C' *is well-typed.*

We don't prove this lemma and instead we will prove an extended version in Section 3.4.

2.5 Verification of safety for μMIR programs

To verify μMIR code, we translate it to a functional language. Then we prove that when the translation does not get stuck, neither does the μMIR program. This might seem redundant since Section 2.4 already proved preservation of typing, but by translating to a functional language it becomes possible to leverage existing deductive verification tools. To verify μMIR code, we will translate it to a functional language and use existing tools to verify the translated program. This approach is also easily extensible to arbitrary specifications which can be represented as pure logical functions.

Simplification In the following sections, we only consider programs without function calls, and therefore we ignore the stacks of the μMIR machine. The problems raised by function calls are orthogonal to the primary challenge of this proof and the one in Section 3.5.3. As a result, programs only consists in the body of the `main` function.

2.5.1 The μML language

For μMIR, we will target a simple ML family language which we call μML. It is an untyped ML dialect equipped with assertions, and call-by-value semantics described by an abstract machine with environment. The rules of the μML abstract machine can be found in Appendix E.

⟨*Expression, e*⟩ ::= '`let`' x '`=`' ⟨*e'*⟩ '`in`' ⟨*e*⟩ | ⟨*e*⟩ ⟨*e'*⟩ | x | ⟨*v*⟩ | ⟨*e*⟩ ⟨*op*⟩ ⟨*e'*⟩ | '`match`' ⟨*e*⟩ '`with`' '`|`' inj_0 x_0 '`->`' ⟨*e*⟩ '`|`' inj_1 x_1 '`->`' ⟨*e*⟩ '`end`' | '`let`' '`(`' x, y '`)`' = ⟨*e*⟩ | '`assert`' '`{`' ⟨*e*⟩ '`}`' | '`(`' ⟨*e*⟩ '`,`' ⟨*e*⟩ '`)`' | inj_i ⟨*e*⟩ | '`rec`' ⟨*FunDef*⟩ '`and`' ... '`and`' ⟨*FunDef*⟩

⟨*FunDef*⟩ ::= '`fun`' f x ... x = ⟨*Expr*⟩

⟨*Values, v*⟩ ::= '`(`' ⟨*v*⟩ '`,`' ⟨*v*⟩ '`)`' | inj_i ⟨*v*⟩ | *n* | '`rec`' ⟨*FunDef*⟩ '`and`' ... '`and`' ⟨*FunDef*⟩

2.5.2 Translating from μMIR to μML

The translation takes a well-typed μMIR CFG and produces a set of mutually recursive μML functions with the same entrypoint as μMIR. A labeled statement becomes a function where the arguments are the entire domain of the partial variable context associated with the label. Each `goto` ℓ is compiled to a function call, and the arguments are the variables in the domain of Ξ_ℓ. Most instructions from μMIR are translated to their direct counterparts in μML. For example, here is the traduction of the construction of a pair:

$$[\![\ell\texttt{: } x \texttt{ := } (y, z)\texttt{; goto } \ell'\texttt{'}]\!] \quad \triangleq \quad \texttt{fun } \ell\ \vec{a} = \texttt{let } x = (y, z) \texttt{ in } \ell'\ \vec{a'}$$

where $\vec{a} = dom(\Xi_\ell)$ and $\vec{a'} = dom(\Xi_{\ell'})$

Interestingly, the operations related to the `box` type are entirely erased since we are not interested in the memory layout in μML:

$$[\![\ell\texttt{: } x^{\texttt{box } T} \texttt{:= box } y^T\texttt{; goto } \ell'\texttt{'}]\!] \quad \triangleq \quad \texttt{fun } \ell\ \vec{a} = \texttt{let } x = y \texttt{ in } \ell'\ \vec{a}$$
$$[\![\ell\texttt{: } x^T \texttt{:= unbox } y^{\texttt{box } T}\texttt{; goto } \ell'\texttt{'}]\!] \quad \triangleq \quad \texttt{fun } \ell\ \vec{a} = \texttt{let } x = y \texttt{ in } \ell'\ \vec{a}$$

where $\vec{a} = dom(\Xi_{\ell'})$.

The full translation rules are presented in Appendix C.

Translating Figure 6 When we run the translation we produce the following set of functions.

```
let rec l0 () = let t1 = 1 in let t2 = 2 in
  let t3 = (t1, t2) in let x = box t3 in l4 x
and l4 x = let t4 = x in let (t5, t6) = t4 in
  let t7 = t6 + 1 in l7 t5 t7
and l7 t5 t7 = let t8 = (t5, t7) in let x = t8 in l9 x
and l9 x = let t9 = x in let (x1, x2) = t9 in l11 x1 x2
and l11 x1 x2 = let t10 = 3 in let t11 = (x2 = t10) in
  assert { t11 }; l14 x1 t10 t11
and l14 x1 t10 t11 = let ret = () in ret
```

2.5.3 Correctness of the translation

We have a translation, but what does it actually produce? To ensure the translation is correct, we will show that whenever the μML translation of a program is *safe*, that is, it does not get stuck, then so is the original program.

Theorem 2.5 (Safety)**.** *Given a well-typed program $\Gamma \vdash \mathcal{P}$, if $[\![\mathcal{P}]\!]$ is safe, then $\mathcal{P}$ is safe.*

In order to prove Theorem 2.5, we establish a simulation $\sim_{\mathcal{P}}$ between the input program $\mathcal{P}$ and its translation $[\![\mathcal{P}]\!]$. We will give the exact definition of $\sim_{\mathcal{P}}$ later, but what's important is that it restricts the μMIR heap and μML environment to ensure that the values in the environment correspond to regions of heap memory. Using this simulation we prove auxiliary lemmas, from which the proof to Theorem 2.5 will be formed.

The proof of safety is achieved using the following three lemmas.

Lemma 2.6 (Preservation of Simulation)**.** *Given a μMIR configuration C and a μML configuration K such that $C \sim_{\mathcal{P}} K$, if $C \to_{\mathcal{P}} C'$ then $K \to K'$ and $C' \sim_{\mathcal{P}} K'$.*

Lemma 2.7 (Progress)**.** *Given a μMIR configuration C and a μML configuration K such that $C \sim_{\mathcal{P}} K$, if K is not stuck then C is not stuck.*

Lemma 2.8 (Terminal Configurations)**.** *Given a terminal μML configuration K, for any μMIR configurations C such that $C \sim_{\mathcal{P}} K$, then C is terminal.*

Proof of Theorem 2.5. Suppose that C is not safe, therefore there exists a trace $C \to_{\mathcal{P}}^{*} C'$ which gets stuck. By iterating Lemma 2.6, there exists K', such that $C' \sim_{\mathcal{P}} K'$. Because K' is safe, it cannot be stuck and must either be terminal or continue reducing. If K' is a terminal configuration then by Lemma 2.8 there is a contradiction since C' is not terminal. If K' is not terminal then there exists K'' such that $K' \to_K K''$ and by Lemma 2.7 there exists C'' such that $C' \to_{\mathcal{P}} C''$, therefore C' is not stuck. □

2.5.4 Simulation Invariant

In order to describe the simulation relation $\sim_{\mathcal{P}}$ which relates μMIR and μML programs, we show how to translate a region of a heap into a μML value. This relation is called the *readback* of the heap. The region of memory associated with each μMIR variable will be translated to a μML value. Because μMIR features only owned values and no borrowing, heap regions are intrinsically separated, which makes the readback a fairly direct and natural operation.

The readback of a heap region is guided by the type of the variable, it extends the heap typing relation covered in Section 2.4, associating a μMIR value to the concerned region of memory.

Definition 2.9 (Readback)**.** *$\mathcal{R}(\mathbf{H}, a, T, v)$ is a 4-place relation between a heap, address, type and a μML value.*

$$\frac{\mathcal{R}(\mathbf{H}_1, a, T_1, v_1) \qquad \mathcal{R}(\mathbf{H}_2, a + |T_1|, T_2, v_2) \qquad \mathbf{H} = \mathbf{H}_1 \oplus \mathbf{H}_2}{\mathcal{R}(\mathbf{H}, a, T_1 \times T_2, (v_1, v_2))}$$

$$\frac{\mathcal{R}(\mathbf{H}, a+1, T_i, v)}{\mathcal{R}(\mathbf{H} \oplus \{(a, i)\}, a, T_0 + T_1, inj_i\ v)} \qquad \frac{\mathcal{R}(\mathbf{H}, p, T, v)}{\mathcal{R}(\mathbf{H} \oplus \{(a, p)\}, a, \texttt{box}\ T, v)}$$

$$\frac{c \textit{ is a literal of } T \qquad T \in \{\texttt{int}, \texttt{bool}, \texttt{unit}\}}{\mathcal{R}(\{(a, c)\}, a, T, c)}$$

The correspondence between a heap, frame and environment is given by HeapEnv which selects the correct elements from the readback to populate the environment.

Definition 2.10 (HeapEnv)**.** HeapEnv$(\mathbf{F}, \mathbf{H}, \mathbf{\Gamma}, E)$ *is a 4-place relation between a μMIR frame and heap, a partial variable environment and a μMIR environment where:*

$$dom(\mathbf{F}) = dom(E) \qquad \mathbf{H} = \bigoplus_{x \in dom(\mathbf{F})} \mathbf{H}_x$$

$$\forall x \in dom(\mathbf{F}), \mathcal{R}(\mathbf{H}_x, \mathbf{F}(x), \mathbf{\Gamma}(x), E(x))$$

The final simulation relation $\sim_{\mathcal{P}}$ uses HeapEnv to relate a μMIR heap to a μML environment and restricts μML programs to the translation of the corresponding μMIR label. The full relation is given in Appendix D along with the proof of Theorem 2.4. The proof proceeds by case analysis on the reductions of μMIR, shuffling memory around to show that the resulting readback is exactly what was expected. We exclude it for space considerations.

2.6 Recap

In Section 2 we presented a small unstructured language with an ownership discipline enforced by typing. This language simulates the behavior of owned values in MIR. We then showed how to translate μMIR programs to μML, a standard ML dialect and proved Theorem 2.5 showing that this translation is correct and permits the verification of μMIR programs.

⟨*Instruction*⟩ ::= ⟨v⟩ '`:=`' '`&mut`'$_\alpha$ ⟨v⟩ | '`immut`' ⟨v⟩ | ⟨v⟩ '`:=`' '`unnest`' ⟨v⟩
| '`(`' '`ref`' ⟨v⟩ '`,`' '`ref`' ⟨v⟩ '`)`' '`:=`' *⟨v⟩ | '`thaw`' α | $\alpha \leq \alpha$

⟨*Statement*⟩ ::= '`match`' *x '`{`' inj$_0$ '`ref` 'y_0 → '`goto`' ℓ_0, inj$_1$ '`ref`' y_1 → '`goto`' ℓ_1, '`}`' | ...

⟨*Type, T*⟩ ::= '`&mut`'$_\alpha$ ⟨*Type*⟩ | '`&`'$_\alpha$ ⟨*Type*⟩ | ...

⟨*Signature*⟩ ::= '`fn`' name '`<`' α, ... | $\alpha \leq \alpha$, ... '`>`' '`(`' ⟨v⟩ '`,`' ... '`)`' '`->`' ⟨*Type*⟩

Figure 7: The syntax of MiniMir as an extension of μMIR

3 Support for borrows and lifetimes

During function calls in μMIR every argument is consumed, making it impossible to call a function with the same argument twice. To solve this, Rust uses *borrows* which allow a variable to lend its read and write permissions temporarily. While a value is borrowed, the original owner is *frozen* until the end of the borrow's *lifetime*. We can pass a function a *borrowed reference* as an argument, giving it temporary ownership of the contents, while recovering control when the borrow expires. We extend μMIR with the operations required to create and manipulate borrows as well as *ghost* operations used to ensure their safety. We call this resulting language MiniMir, because it's a mini language that does the maximum! [1]

3.1 The MiniMir language

Just like in μMIR, programs are collections of function declarations. The syntax of signatures is more complex, functions can be parameterized over lifetime variables α and lifetime constraints $\alpha \leq \beta$. Let us consider an example:

`fn fst_proj <` α `| >` `(p :` `&mut`$_\alpha$`int x int)` → `&mut`$_\alpha$`int`

The signature for `fst_proj` takes as input a mutable borrow alive for α of a pair of ints (represented at runtime as a pointer to a pair of ints), and returns a borrow lasting for the same lifetime α of a single int. Here, function is parameterized over α, allowing us to instantiate it with different lifetimes at different points in a program.

The instructions and statements are extended with operations to create a mutable borrow ($x^{\texttt{\&mut}_\alpha\, T}$`:= &mut`$_\alpha y^T$), to turn a mutable borrow into an immutable borrow (`immut` $y^{\texttt{\&mut}_\alpha\, T}$) and to *unnest* borrows ($x^{\texttt{\&mut}_\alpha\, T}$`:= unnest` $y^{\texttt{\&mut}_\alpha\, P\, T}$). Unnesting is an essential operation when working with borrows. It collapses a layer of indirection between borrows, transforming a `&mut`$_\alpha$`&mut`$_\beta$ T into a `&mut`$_\alpha$ T. To interact with product types, a borrow of a pair can be destructed into a pair of borrows (`let (ref` $x^{P\, T_0}$`, ref` $y^{P\, T_1}$`) :=` $*z^{P\,(T_0 \times T_1)}$).

The statements of MiniMir includes a new kind of match (`match *x { .. }`) which allows programs to turn a borrow on a sum into a borrow on the value contained in the sum. When this occurs, programs are free to modify the value held in the sum but cannot change the constructor of the sum.

To verify the safety of borrows, we also add several ghost instructions, to thaw (end) a lifetime and to impose an ordering over lifetimes.

We can see this translation applied to Figure 4 in Figure 8. In this translation we inlined the `inc` function to make presentation simpler.

[1] Reference to the famous slogan for a French cleaning product "Mini Mir, Mini Prix, mais il fait le maximum" **reference needed**

```
ℓ1: t1 := 0; t2 := &mut_α t1; goto ℓ3

ℓ3: t3 := *t2; t4 := t3 + 1; t5 := &mut_α t4; swap(t2, t5); goto ℓ7
ℓ7: immut t5; drop t5; goto ℓ9

ℓ9: t6 := *t2; t7 := t6 + 1; t8 := &mut_α t7; swap(t2, t8); goto ℓ13
ℓ13: immut t8; drop t8; goto ℓ15

ℓ15: immut t2; drop t2; thaw α; goto ℓ17
ℓ17: t4 := t1; assert { t4 = 2 }; drop t4; t0 := (); drop t1; return t0
```

Figure 8: Translation of Figure 4, with the function `inc` inlined.

3.2 Extending types with borrowing

By creating pointers to values, borrows introduce aliasing. To prevent that, when a value is borrowed for a lifetime α, the original name will be made inaccessible or frozen until the end of α. To verify this safety property, the type system tracks a partial order on lifetimes, ensuring that borrows cannot outlive their prescribed lifetime. When the type system encounters a `thaw` α instruction, it checks that all relevant borrows have been released and restores access to frozen variables.

We extend the type system of μMIR with *partial lifetime contexts*, which consist of a collection of elements $\alpha \leq \beta$. The partial variable context for MiniMir consists of elements of the form $x :^{\dagger\alpha} T$ or $x :^{\bullet} T$, the first denoting a variable is frozen for lifetime α while the second denotes it is not frozen.

The whole context of a function $(\mathbf{\Xi}, \mathbf{\Lambda})$ also collects the partial lifetime contexts for every label in the function. The forms of judgements extend naturally, using the lifetime contexts where appropriate. The judgements for instructions and statements are extended with partial lifetime contexts, while the judgement for functions is extended with a whole lifetime context.

The typing judgement for $x^{\texttt{\&mut}_\alpha\, T} \texttt{:= \&mut}_\alpha y^T$ expresses how the original value y is frozen until α expires while granting x access for that lifetime. No modifications are made to the partial lifetime context $\mathbf{L}$ as creating a borrow does not immediately place it in an ordered relation.

$$\frac{}{\mathbf{\Sigma}; y :^{\bullet} T, \mathbf{\Gamma}; \mathbf{L} \vdash_f x^{\texttt{\&mut}_\alpha\, T} \texttt{:= \&mut}_\alpha y^T \dashv \mathbf{\Gamma}, x :^{\bullet} \texttt{\&mut}_\alpha\, T, y :^{\dagger\alpha} T; \mathbf{L}} \text{(BORROW-MUT)}$$

When a lifetime is ended using a `thaw` α, the type system ensures that everything borrowed for α has been dropped and that all lifetimes $\beta \leq \alpha$ have already been thawed. After a lifetime is thawed, all variables in the context which were frozen for α are unfrozen, restoring access. The complete type system for MiniMir is included in Appendix F.

3.3 Operational Semantics of MiniMir

The operational semantics of MiniMir changed little compared to μMIR. Because instructions like `thaw` are ghost, they become no-ops in the semantics. The primary addition of MiniMir, the mutable borrow, creates a new pointer to a value. The extended semantics for MiniMir is included in Appendix G.

3.4 Type preservation by reduction

Simplification In the following sections, as in the end of Section 2, we only consider programs without function calls. Again, the problems raised by function calls are orthogonal to the primary

challenge of this proof and the one in Section 3.5.3.

The banality of the semantics for mutable references is exactly what we desired. They create pointers that the type system ensures can only be used safely. To prove this we extend Theorem 2.4 to the full language of MiniMir.

We extend the heap fragment typing of Section 2.4 to include reference types. Since mutable references are non-aliasing, the heap fragment types must ensure that there are never two active pointers to the same region of memory. To ensure mutable borrows are only used once in a heap, we use a *borrow store* The borrow store holds tokens for the lender and the borrower. By consuming the borrow store, the heap typing can ensure each borrow is used only once.

Definition 3.1 (Borrow Store)**.** *A* borrow store *is a finite set of elements of the form* $\mathbf{take}^r_\alpha(a, T)$ *or* $\mathbf{give}^r_\alpha(a, T)$*, where* a *is an address,* T *is a type and* r *is either* i *for immutable or* m *for mutable.*

A borrow store $\mathcal{B}$ *is* safe *if for every address* a *appearing in* $\mathcal{B}$ *it contains exactly the tokens* $\mathbf{take}^m_\alpha(a, T)$ *and* $\mathbf{give}^m_\alpha(a, T)$ *or contains* $\mathbf{give}^i_\alpha(a, T)$ *and zero or more* $\mathbf{take}^i_\alpha(a, T)$*.*

We extend heap fragment typing and well-typing of Section 2.4. The judgments have the form, $\mathcal{B}; \mathbf{H} \models a :^{\mathbf{n}} T$, where $\mathcal{B}$ a borrow store and $\mathbf{n} \in \{\dagger\alpha; \bullet\}$. The definition of a well-typed configuration is then extended to ensure borrow stores are safe.

Definition 3.2 (Heap Fragment Type for MiniMir)**.** *Below are the heap typing rules for mutable and immutable references. The borrow store checks that both the lender and borrower agree on the addresses and types borrowed. The full rules are found in Appendix I.*

$$\frac{}{\mathbf{give}^m_\alpha(a, T); \varnothing \models a :^{\dagger\alpha} T} \qquad \frac{\mathcal{B}; \mathbf{H} \models a :^{\dagger\alpha} T}{\mathbf{give}^i_\alpha(a, T), \mathcal{B}; \mathbf{H} \models a :^{\dagger\alpha} T}$$

$$\frac{\mathcal{B}; \mathbf{H} \models a :^{\mathbf{n}} T}{\mathbf{take}^m_\alpha(a, T), \mathcal{B}; \{(p, a)\} \oplus \mathbf{H} \models p :^{\mathbf{n}} \texttt{\&mut}_\alpha T} \qquad \frac{}{\mathbf{take}^i_\alpha(a, T); \{(p, a)\} \models p :^{\mathbf{n}} \texttt{\&}_\alpha T}$$

Definition 3.3. *A well-typed configuration* $\langle f; \ell \mid - \mid \mathbf{F} \mid \mathbf{H} \rangle$ *of a well-typed program* $\mathcal{P}$ *if it satisfies the following conditions:*

$$dom(\Xi_\ell) = dom(\mathbf{F}) \qquad \mathbf{H} = \bigoplus_{x \in \mathbf{F}} \mathbf{H}_x \qquad \mathcal{B} = \bigoplus_{x \in \mathbf{F}} \mathcal{B}_x \qquad safe(\mathcal{B})$$

$$\mathcal{B}_x; \mathbf{H}_x \models x :^{\mathbf{n}} \Xi_\ell(x, \mathbf{n})$$

Using these expanded definitions, the proof of preservation extends easily to handle mutable and immutable references. Whenever a borrow is created or dropped, the corresponding tokens are inserted or removed from the borrow store.

Theorem 3.4 (Type Preservation)**.** *Given a well-typed MiniMir configuration* C*, if* $C \to_{\mathcal{P}} C'$*, then* C' *is well-typed.*

The interesting cases of this proof are discussed in Appendix I

3.5 Verification by translation

We extend our translation from μMIR with support for mutable and immutable references. Our encoding of mutable borrows into a functional language relies on their non-aliasing. When a borrow is created through $x^{\texttt{\&mut}_\alpha\, T} \texttt{:= \&mut}_\alpha y^T$, the type safety of MiniMir tells us that y cannot

```
fun l1 () = let t1 = 0 in let t2 = (t1, any) in let t1 = ^t2 in l3 t1 t2
and fun l3 t1 t2 = let t3 = *t2 in let t4 = t3 + 1 in
  let t5 = (t4, any) in let t4 = ^ t5 in let temp = *t2 in
  let t2 = ( *t5, ^t2) in let t5 = (temp, ^t5) in l7 t1 t2 t3

and fun l7 t1 t2 t5 = assume { * t5 = ^ t5}; l9 t1 t2

and fun l9 t1 t2 = let t6 = *t2 in let t7 = t6 + 1 in
  let t8 = (t7, any) in let t7 = ^t8 in let temp = *t2 in
  let t2 = ( *t8, ^t2) in let t8 = (temp, ^t8) in l13 t1 t2 t8

and fun l13 t1 t2 t8 = assume { * t8 =  ^ t8 }; l15 t1 t2

and fun l15 t1 t2 = assume { * t2 = ^ t2 }; l17 t1
and fun l17 t1 = let t4 = t1 in assert { t4 = 2}; return ()
```

Figure 9: Translation of Figure 8 to MiniML

be used until the end of α. At the end of α, y will have been updated with all the changes performed on x. In some sense, when we translate $x^{\texttt{\&mut}_\alpha\ T}\texttt{:= \&mut}_\alpha y^T$, we would like to replace y with the final value pointed to by x. Since we can't see into the future, we non-deterministically guess a value for y. Our translation represents mutable borrows as a *pair* of the current and final value being borrowed, like in RustHorn[12]. When the borrow is created we assign to y the *final* value of the borrow. As the translated program executes, the current value of x is updated. When the borrow is frozen, we rule out any executions that guessed the wrong final value by checking that the current and final values are equal.

3.5.1 The MiniML language

MiniML is an extension of μML with non-determinism. The `any` expression picks an arbitrary value, and a `assume { e }` evaluates an assumption `e`, if it doesn't reduce to `true`, then it diverges. The operational semantics of μML are extended to include these constructs in Appendix J.

⟨*Expression, e*⟩ ::= '`any`' | '`assume`' '`{`' e '`}`' | ...

3.5.2 Translating from MiniMir to MiniML

The translation of μMIR to μML is adapted to MiniMir. When a borrow is created, the borrow is assigned the current value being borrowed and its final value. In the translation we use two operators to access borrows defined as: $\texttt{*}x \triangleq \texttt{fst}\ x$ and $\texttt{\^{}}x \triangleq \texttt{snd}\ x$.

$$[\![\ell\texttt{: }x^{\texttt{\&mut}_\alpha\ T}\texttt{:= \&mut}_\alpha y^T\texttt{; goto }\ell\texttt{'}]\!] \triangleq \begin{array}{l}\texttt{fun }\ell\ \vec{a}\texttt{ = let }x\texttt{ = (}y\texttt{, any) in}\\ \texttt{let }y\texttt{ = \^{}}x\texttt{ in }\ell'\ \vec{a'}\end{array}$$

When a mutable borrow is made immutable, we use the `assume` expression to equate its current value and final value.

$$[\![\ell\texttt{: immut }y^{\texttt{\&mut}_\alpha\ T}\texttt{; goto }\ell\texttt{'}]\!] \triangleq \texttt{fun }\ell\ \vec{a}\texttt{ = assume \{ *}y\texttt{ = \^{}}y\texttt{\}; }\ell'\ \vec{a'}$$

The full translation is presented in Appendix H. In Figure 9, the program of Figure 8 is translated to MiniML. Instructions with produce no output in the translation are elided entirely.

3.5.3 Correctness of translation

We now examine the correctness of the translation presented in the previous section. Our safety theorem has the same structure as in μMIR but because of the non-determinism of MiniML the notion of safety changes to be programs in which all traces are non blocking.

Theorem 3.5 (Safety)**.** *Given a well-typed MiniMir program $\vdash \mathcal{P}$, if $[\![\mathcal{P}]\!]$ is safe, then $\mathcal{P}$ is safe.*

The proof of this theorem is structured in the same manner as for μMIR. To establish a simulation between MiniMir and MiniML we must extend the translation of memory to handle borrows.

When we define the simulation between MiniMir and MiniML, we find that mutable borrows cause us a problem. Each time a borrow is created, the translated program must guess the final value of the borrow, a value which appears nowhere in the memory of the MiniMir program. The readback must make the correct guess to constrain the traces in the simulation. We simplify this problem by supposing the existence of a *prophecy map*, which contains the correct final value for every pair of borrowed address and type. Using this prophecy map, we can define the readback of MiniMir as an extension of the heap typing just like with μMIR.

Definition 3.6 (Readback)**.** *The readback of Definition 2.9 is extended with support for mutable and immutable borrows. For each value being readback, it will produce a mapping containing every sub-value within.*

$$\frac{A(a,T) = v}{\mathcal{R}^{\dagger\alpha}(\{\mathbf{give}^m_\alpha(a,T)\}, A, \varnothing, a, T, (a,T,v))} \qquad \frac{\mathcal{R}^{\dagger\alpha}(\mathcal{B}, A, \mathbf{H}, a, T, E)}{\mathcal{R}^{\dagger\alpha}(\{\mathbf{give}^i_\alpha(a,T)\} \cup \mathcal{B}, A, \mathbf{H}, a, T, E)}$$

$$\frac{A(a,T) = v}{\mathcal{R}^{\mathbf{n}}(\{\mathbf{take}^i_\alpha(a,T)\}, A, \{(p,a)\}, p, \texttt{\&}_\alpha\ T, (p,T,v))}$$

$$\frac{\mathcal{R}^{\mathbf{n}}(\mathcal{B}, A, \{a\} \lhd \mathbf{H}, \mathbf{H}(a), T, E) \qquad A(\mathbf{H}(a), T) = v}{\mathcal{R}^{\mathbf{n}}(\mathbf{take}^m_\alpha(a,T) \cup \mathcal{B}, A, \{(p,a)\} \oplus \mathbf{H}, p, \texttt{\&mut}_\alpha\, T, E + (a, T \times T, (E(\mathbf{H}(a)), v)))}$$

Definition 3.7 (HeapEnv)**.** *The relation* HeapEnv *shows how to construct an ML environment from a MiniMir configuration. It uses the heap types of a configuration to readback memory cells as ML values.*

Formally, HeapEnv $(\mathcal{B}, A, \mathbf{F}, \mathbf{H}, \mathbf{\Gamma}, E)$ *is a 6-place relation between an activeness, a borrow store, a prophecy map, a MiniMir frame and heap, a partial typing environment and a MiniML environment where:*

$$dom(\mathbf{F}) = dom(E) \qquad \mathbf{H} = \bigoplus_{x \in dom(\mathbf{F})} \mathbf{H}_x \qquad E = \{(x, V_x(F(x), T)) \mid x :^{\mathbf{n}} T \in \mathbf{\Gamma}\}$$

$$\forall x \in dom(\mathbf{F}), \mathcal{R}^{\mathbf{n}}(\mathcal{B}_x, A, \mathbf{H}_x, \mathbf{F}(x), \mathbf{\Gamma}(x, \mathbf{n}), V_x)$$

It turns out, rather essentially, that the prophecy map we need can be calculated from a MiniMir trace. To do this, we take a trace and walk it backwards, each time a `thaw` α is encountered, the end of a lifetime has been reached. At that moment the values of all variables that were frozen are readback.

$$\frac{\mathcal{P}_{f;l} = \texttt{thaw}\ \alpha \qquad A' = A \oplus \bigoplus_{\mathbf{give}^r_\alpha(a,T) \in \mathcal{B}} \{E \mid \mathcal{R}^{\dagger\alpha}(\mathcal{B}'_a, A, \mathbf{H}'_a, a, \mathbf{\Xi}_{f';\ell'}(a, \bullet), E)\}}{\text{McFly}^*(A', \langle [f;\ell] \mid - \mid \mathbf{F} \mid \mathbf{H} \rangle \to_{\mathcal{P}} \langle f';\ell' \mid - \mid \mathbf{F}' \mid \mathbf{H}' \mid \mathcal{B}' \rangle, A)} \quad \text{HeapEnv}\ (A, \mathcal{B}, \mathbf{F}', \mathbf{H}', \mathbf{\Xi}_{f;\ell}, E)$$

In order to use McFly* we use a simulation between MiniMir traces and MiniML configurations. The simulation relation, $\sim_{\mathcal{P}}$ calculates the required prophecy map using McFly*, and uses it to ensure that the heap translates to the MiniML environment. At the same time the code is constrained to be the translation of the initial configuration of the trace.

We structure the proof of Theorem 3.5 using the same three lemmas as before: progress, terminal configurations and preservation. Using these lemmas, the proof proceeds in the same manner as Theorem 2.5. The statements of the first two lemmas change to account for the non-determinism of MiniML but, the structure of their proofs remains the same, thus we leave them in Appendix I, and focus on the proof of Lemma 3.8.

Lemma 3.8 (Preservation of Simulation)**.** *Given a MiniMir trace $\Theta = C \to^*_{\mathcal{P}} C'$ and a MiniML configuration K such that $\Theta \sim_{\mathcal{P}} K$, if $C \to_{\mathcal{P}} C''$, there exists a K' such that $K \to K'$ and $C'' \to^*_{\mathcal{P}} C' \sim_{\mathcal{P}} K'$.*

The general structure of the proof remains unchanged from μMIR, it proceeds by case analysis on the reductions of MiniMir. The reductions relating to owned values proceed exactly as before, shuffling information around. Instead, the essential difficulty of this proof comes down to showing that McFly* finds the correct final value for every borrow. The intuition is that once a value is frozen, it cannot change, and therefore at the moment that we find the prophecy it must have the same value as when it was frozen.

Consider the case of $\texttt{immut}\ y^{\texttt{\&mut}_\alpha\ T}$, here we freeze a mutable reference `x`, the translation of this gives $\texttt{assume \{ *}y\texttt{ = \^{}}y\texttt{\};}\ \ell\ \vec{a}$. To preserve the simulation, the MiniMir configuration must reduce to $\ell'\ \vec{a}$, with memory compatible with the heap. In sum, this means showing that the final value of x is the same as it's current value.

By the preservation of heap typing, we observe that when the borrow is frozen, the memory $\mathbf{H}_x$ of x must be transferred back to the variable which was borrowed. To show that McFly* found the correct final value, we note that by inversion on McFly*, the prophecy for x must come from some future configuration C'' of C. We then use Lemma I.6 to show that the memory of C'' can be readout. Finally, since by typing frozen cells of memory cannot change by reduction, the readout of $\mathbf{H}_x$ must be the same at C'' as it was at C. Since this value forms the prophecy for x, it must be the case the the current and final values of the borrow are equal and that the MiniML program can progress.

The proof is detailed more formally in Appendix I.

4 Experimentation

As further validation of the translation presented in Section 3, we implemented a proof-of-concept tool as an extension to the Rust compiler. This tool takes Rust programs in their MIR representation and translates them to WhyML, the specification and programming language of the Why3 verification suite. The translated programs can then be checked using the Why3 prover interface. Using this tool we were able to verify simple programs working on structures such as linked lists.

One of the primary implementation challenges is determining where the `thaw` should be in MIR code. The borrow checker of Rust infers a position where it should be inserted but that information is hidden from other passes. Extracting that information is essential as the safety of this tool relies on correct placement of thaws to mark the end of lifetimes.

We also took advantage of this tool to test several simple extensions, such as support for pre-conditions, post-conditions and invariants in code being verified. Each of these is lowered, nearly directly to the equivalent Why3 constructs. This enabled us to verify the functional correctness of several programs. A few of these programs are included in Appendix K

5 Conclusion

During this internship, we developed a schema to verify Rust-like programs by translation to a functional language. Our source language MiniMir, is translated to MiniML, an ML-like language with non-determinism. In our translation we represent mutable borrows as pairs of their current and future values. To prove that our translation is correct, we developed an original simulation technique between an execution trace and a functional configuration. The proof requires us to show that we can predict the future values of each borrow, and we show that we can statically identify points in the program which can be used to predict those values. Finally, we validated this technique experimentally by implementing this translation as a proof-of-concept tool. Our tool was able to verify safety properties for simple programs using list operations.

Related Work. The Frama-C[9] tool for C allows the verification of C in the presence of aliasing through non-aliasing hypotheses which assert pair-wise non-aliasing of variables. Asserting that all variables are non-aliasing requires quadratic amounts of hypotheses, which overwhelms automation like SMT solvers. Other languages like Spark/Ada rule out all aliasing through their type system. Recent work has been done on how to add aliasing references like in Rust[6].

To verify Rust programs, the team behind Prusti[4] translates programs to a separation logic with fractional permissions. Currently, their approach is limited in the properties it is able to prove and their implementation supports a more limited subset of Rust programs than the approach offered by MiniMir. Another approach is that of RustHorn[12], which translates programs to Constrained Horn Clauses using the same encoding of mutable references we use. RustHorn however cannot represent user specifications and can only show that programs don't fail assertions. Additionally, their approach of translating to CHCs means that they rely entirely on automated solvers with no method to allow human guidance.

Future Work. The approach presented in section 3 allows us to verify Rust-style programs using both mutable and immutable references. However, the proof of correctness is complex using an original form of simulation. Formalizing this development in Coq would give a much firmer base for further extensions to the language. The work on RustBelt[8], could serve a starting point for a Coq formalization, since key lemmas like type safety are already proven on the core language λRust.

Currently, we have not considered the questions of specification languages for Rust, the encoding for datatype invariants and ghost code more generally remains an open question. To put this translation into application and incorporate extensions, the tool developed for this internship needs to be extended to more gracefully handle real-world Rust programs.

References

[1] Rust Programming Language. https://www.rust-lang.org/.

[2] Announcing Rust 1.0. https://blog.rust-lang.org/2015/05/15/Rust-1.0.html, May 2015.

[3] Aliasing (computing). https://en.wikipedia.org/w/index.php?title=Aliasing_(computing)&oldid=959584508, May 2020.

[4] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. Leveraging Rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, October 2019.

[5] Richard Bornat. Proving Pointer Programs in Hoare Logic. In Roland Backhouse and José Nuno Oliveira, editors, *Mathematics of Program Construction*, Lecture Notes in Computer Science, pages 102–126, Berlin, Heidelberg, 2000. Springer.

[6] Claire Dross and Johannes Kanig. Recursive Data Structures in SPARK. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 178–189, Cham, 2020. Springer International Publishing.

[7] Patrice Godefroid. Software model checking: The VeriSoft approach. *Formal Methods in System Design*, 26(2):77–101, 2005.

[8] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, January 2018.

[9] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, May 2015.

[10] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert-a formally verified optimizing compiler. 2016.

[11] Nicholas D. Matsakis and Felix S. Klock. The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, October 2014.

[12] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. RustHorn: CHC-based Verification for Rust Programs (full version). In *ESOP*, page 49, 2020.

A Complete Type System of μMIR

$$\frac{}{Copy(\texttt{int})} \qquad \frac{}{Copy(\texttt{bool})} \qquad \frac{}{Copy(\texttt{unit})}$$

$$\frac{Copy(T_0) \quad Copy(T_1)}{Copy(T_0 \times T_1)} \qquad \frac{Copy(T_0) \quad Copy(T_1)}{Copy(T_0 + T_1)}$$

$$\frac{y : T \in \mathbf{\Gamma} \quad Copy(T)}{\mathbf{\Sigma}; \mathbf{\Gamma} \vdash_f x^T \texttt{:= copy } y^T \dashv \mathbf{\Gamma}, x : T} \text{ (COPY-VAL)}$$

$$\frac{y : \texttt{box } T \in \mathbf{\Gamma} \quad Copy(T)}{\mathbf{\Sigma}; \mathbf{\Gamma} \vdash_f x^T \texttt{:= copy *}y^{P\,T} \dashv \mathbf{\Gamma}, x : T} \text{ (COPY-REF)}$$

$$\frac{}{\mathbf{\Sigma}; y : \texttt{box } T, \mathbf{\Gamma} \vdash_f x^T \texttt{:= unbox } y^{\texttt{box}\,T} \dashv \mathbf{\Gamma}, x : T} \text{ (UNBOX)}$$

$$\frac{}{\mathbf{\Sigma}; y : T, \mathbf{\Gamma} \vdash_f x^{\texttt{box}\,T} \texttt{:= box } y^T \dashv \mathbf{\Gamma}, x : \texttt{box } T} \text{ (BOX)}$$

$$\frac{x : \texttt{box}\,T, y : \texttt{box}\,T \in \mathbf{\Gamma}}{\mathbf{\Sigma}; \mathbf{\Gamma} \vdash_f \texttt{swap(}x^{P\,T}\texttt{, } y^{P\,T}\texttt{)} \dashv \mathbf{\Gamma}} \text{ (SWAP)} \qquad \frac{T \in \{\texttt{int}, \texttt{unit}\}}{\mathbf{\Sigma}; x : T, \mathbf{\Gamma} \vdash_f \texttt{drop } x^T \dashv \mathbf{\Gamma}} \text{ (DROP)}$$

$$\frac{}{\mathbf{\Sigma}; y : T_i, \mathbf{\Gamma} \vdash_f x^{T_0+T_1} \texttt{:= inj}_i\ y^T \dashv \mathbf{\Gamma}, x : T_0 + T_1} \text{ (INTRO-SUM)}$$

$$\frac{}{\mathbf{\Sigma}; y : T_0, z : T_1, \mathbf{\Gamma} \vdash_f x^{T_0 \times T_1} \texttt{:= (}y^{T_0}\texttt{, } z^{T_1}\texttt{)} \dashv \mathbf{\Gamma}, x : T_0 \times T_1} \text{ (INTRO-PAIR)}$$

$$\frac{}{\mathbf{\Sigma}; \mathbf{\Gamma} \vdash_f \texttt{let x = C} \dashv \mathbf{\Gamma}, x : T_C} \text{ (CONST-INTRO)} \qquad \frac{x : \texttt{bool} \in \mathbf{\Gamma}}{\mathbf{\Sigma}; \mathbf{\Gamma} \vdash_f \texttt{assert x} \dashv \mathbf{\Gamma}} \text{ (ASSERT)}$$

$$\frac{}{\mathbf{\Sigma}; y : \texttt{int}, z : \texttt{int}, \mathbf{\Gamma} \vdash_f x^T \texttt{:= } y^{\texttt{int}}\texttt{op } z^{\texttt{int}} \dashv \mathbf{\Gamma}, x : T_{op}} \text{ (OP)}$$

$$\frac{}{\mathbf{\Sigma}; z : T_0 \times T_1, \mathbf{\Gamma} \vdash_f \texttt{let (}x^{T_0}\texttt{, } y^{T_1}\texttt{) := } z^{T_0 \times T_1} \dashv \mathbf{\Gamma}, x : T_0, y : T_1} \text{ (ELIM-PAIR)}$$

$$\frac{\texttt{fn g } (x_0 : T_0, ...x_{n-1} : T_{n-1}) \to T_n \in \mathbf{\Sigma}}{\mathbf{\Sigma}; x_0 : T_0, .., x_{n-1} : T_{n-1}, \mathbf{\Gamma} \vdash_f \texttt{let x = g(x}_0\texttt{, .., x}_{n-1}\texttt{)} \dashv \mathbf{\Gamma}, x : T_n} \text{ (CALL)}$$

$$\frac{\mathbf{\Sigma}; \mathbf{\Xi}_a \vdash_f I \dashv \mathbf{\Sigma}; \mathbf{\Xi}_\ell}{\mathbf{\Sigma}; \mathbf{\Xi} \vdash_{f,a} \texttt{I; goto } \ell} \text{ (SEQUENCE)} \qquad \frac{\mathbf{\Xi}_a = \{x : \mathbf{\Sigma}_{ret\,f}\}}{\mathbf{\Sigma}; \mathbf{\Xi}_a \vdash_{f,a} \texttt{return x}} \text{ (RETURN)}$$

where $\mathbf{\Sigma}_{ret\ f} = \{T_n \mid \texttt{fn g } (x_0 : T_0, ...x_{n-1} : T_{n-1}) \to T_n = \mathbf{\Sigma}(f)\}$

$$\frac{\mathbf{\Xi}_a = x : T_0 + T_1, \mathbf{\Gamma}_a \quad \mathbf{\Xi}_{\ell_i} = y_i : T_i, \mathbf{\Gamma}_a \quad i \in \{0, 1\}}{\mathbf{\Sigma}; \mathbf{\Xi} \vdash_{f,a} \texttt{match } x^{T_0+T_1}\texttt{\{ inj}_0\ y_0^{T_0} \to \texttt{goto } \ell_0\ \texttt{, inj}_1\ y_1^{T_1} \to \texttt{goto } \ell_1\ \texttt{\}}} \text{ (MATCH-VAL)}$$

$$\frac{\boldsymbol{\Xi}_\ell = \bigcup x_i : T_i \quad \forall S \in \Delta, \boldsymbol{\Sigma}; \boldsymbol{\Xi}; \Delta \vdash_{f,a} S}{\boldsymbol{\Sigma}, \boldsymbol{\Xi} \vdash (\Delta, \ell, \texttt{fn g}\ (x_0 : T_0, ...x_{n-1} : T_{n-1}) \to T_n)}\ (\textsc{function})$$

$$\frac{\boldsymbol{\Sigma} = \{\sigma \mid \forall(\Delta, \ell, \sigma) \in \mathcal{P}\} \quad \forall f \in \mathcal{P}, \boldsymbol{\Sigma} \vdash f}{\vdash \mathcal{P}}\ (\textsc{program})$$

B Complete Operational Semantics of μMIR

Definition B.1 (Notations)**.** *If F is a partial function, and A is a subset of its domain, then $A \triangleleft F$ denotes the* domain restriction *removing A from the domain of F, and is defined as*

$$A \triangleleft F = \{(x, v) \mid (x, v) \in F, x \notin A\}$$

The operation $A \oplus B$ denotes the disjoint union *of partial functions, it is only defined when $dom(A) \cap dom(B) = \varnothing$. The notation $a \perp b$ (*disjoint sets*) signifies that $a \cap b = \varnothing$. The notation $a_0 \perp a_1 \perp .. \perp a_n$ is used to denote the pairwise disjointness of the sets $a_0, .., a_n$. The notation $\mathcal{M}_{\mathbf{H}}(t, s, n)$ (*memory copy*) is a shorthand for copying n cells of $\mathbf{H}$ starting at s to addresses starting at t. It is defined as*

$$\mathcal{M}_{\mathbf{H}}(t, s, n) = \{(t + i, \mathbf{H}(s + i)) \mid i \in [n]\}$$

The notation $[n]$ is used for the index set *$\{i \mid 0 \leq i < n\}$. The notation $[a, b)$ is used for $[b] - [a]$, the set representing the* right-open interval *from a to b*

In a program $\mathcal{P}$, the initial configuration is given by $\langle \texttt{main}; \ell_0 \mid \epsilon \mid \varnothing \mid \varnothing \rangle$, where ℓ_0 is the entrypoint of `main` in $\mathcal{P}$. The terminal configurations of $\mathcal{P}$ are of the form $\langle main; \ell \mid \epsilon \mid \varnothing \mid \varnothing \rangle$, where $\mathcal{P}_{main,\ell} = \texttt{return}\ x$.

The stack has the following form:

$$\mathbf{S} ::= \epsilon \mid [f; \ell, x, \mathbf{F}]; \mathbf{S}$$

As the name implies, it represents the function calls currently being evaluated. Each element of the stack is a triple composed of a return label, a variable name for the result and a frame obtained from the caller at the moment the call is performed.

$$\frac{\mathcal{P}_{f;\ell} = x^T \texttt{:= copy}\ y^T\texttt{; goto}\ \ell\texttt{'} \quad \mathbf{F}(y) = a \quad [b, b + |T|) \perp dom(\mathbf{H})}{\langle f; \ell \mid \mathbf{S} \mid \mathbf{F} \mid \mathbf{H} \rangle \to_{\mathcal{P}} \langle \ell' \mid \mathbf{S} \mid \mathbf{F} + \{(x, b)\} \mid \mathbf{H} + \mathcal{M}_{\mathbf{H}}(b, a, |T|) \rangle}$$

$$\frac{\mathcal{P}_{f;\ell} = x^T \texttt{:= unbox}\ y^{\texttt{box}\,T}\texttt{; goto}\ \ell\texttt{'} \quad \begin{array}{c} [b, b + |T|) \perp dom(\mathbf{H}) \\ A = \{a\} \cup [\mathbf{H}(a), \mathbf{H}(a) + |T|) \end{array}}{\langle f; \ell \mid \mathbf{S} \mid \mathbf{F} + \{(y, a)\} \mid \mathbf{H} \rangle \to_{\mathcal{P}} \langle \ell' \mid \mathbf{S} \mid \mathbf{F} + \{(x, b)\} \mid A \triangleleft \mathbf{H} + \mathcal{M}_{\mathbf{H}}(b, \mathbf{H}(a), |T|) \rangle}$$

$$\frac{\mathcal{P}_{f;\ell} = x^{\texttt{box}\,T} \texttt{:= box}\ y^T\texttt{; goto}\ \ell\texttt{'} \quad \{b\} \perp [c, c + |T|) \perp dom(\mathbf{H}) \quad A = [a, a + |T|)}{\langle f; \ell \mid \mathbf{S} \mid \mathbf{F} + \{(y, a)\} \mid \mathbf{H} \rangle \to_{\mathcal{P}} \langle \ell' \mid \mathbf{S} \mid \mathbf{F} + \{(x, b)\} \mid A \triangleleft \mathbf{H} + \{(b, c)\} + \mathcal{M}_{\mathbf{H}}(c, a, |T|) \rangle}$$

$$\frac{\mathcal{P}_{f;\ell} = \texttt{drop}\ x^T\texttt{; goto}\ \ell\texttt{'}}{\langle f; \ell \mid \mathbf{S} \mid \mathbf{F} + \{(x, a)\} \mid \mathbf{H} \rangle \to_{\mathcal{P}} \langle \ell' \mid \mathbf{S} \mid \mathbf{F} \mid \{a\} \triangleleft \mathbf{H} \rangle}$$

$$\frac{\mathcal{P}_{f;\ell} = \texttt{assert x; goto } \ell\texttt{'} \qquad \mathbf{H}(\mathbf{F}(x)) = \texttt{true}}{\langle f;\ell \mid \mathbf{S} \mid \mathbf{F} \mid \mathbf{H}\rangle \to_{\mathcal{P}} \langle \ell' \mid \mathbf{S} \mid \mathbf{F} \mid \mathbf{H}\rangle}$$

$$\frac{\mathcal{P}_{f;\ell} = \texttt{swap(}x^{P\,T}\texttt{, }y^{P\,T}\texttt{); goto } \ell\texttt{'} \qquad \mathbf{F}(x) = a \qquad \mathbf{F}(y) = b}{\langle f;\ell \mid \mathbf{S} \mid \mathbf{F} \mid \mathbf{H}\rangle \to_{\mathcal{P}} \begin{array}{r}\langle f;\ell \mid \mathbf{S} \mid \mathbf{F} \mid ([\mathbf{H}(a), \mathbf{H}(a)+|T|) \cup [\mathbf{H}(b), \mathbf{H}(b)+|T|)) \triangleleft \mathbf{H} \\ +\mathcal{M}_{\mathbf{H}}(\mathbf{H}(a), \mathbf{H}(b), |T|) + \mathcal{M}_{\mathbf{H}}(\mathbf{H}(b), \mathbf{H}(a), |T|)\rangle\end{array}}$$

$$\frac{\mathcal{P}_{f;\ell} = \texttt{x := C; goto } \ell\texttt{'} \qquad \{a\} \perp dom(\mathbf{H})}{\langle f;\ell \mid \mathbf{S} \mid \mathbf{F} \mid \mathbf{H}\rangle \to_{\mathcal{P}} \langle \ell' \mid \mathbf{S} \mid \mathbf{F} + \{(x,a)\} \mid \mathbf{H} + \{(a,C)\}\rangle}$$

$$\frac{\begin{array}{c} i = H(a) \qquad [b, b+|T_i|) \perp dom(\mathbf{H}) \\ \mathcal{P}_{f;\ell} = \texttt{match } x^{T_0+T_1}\texttt{\{ inj}_0\ y_0^{T_0} \to \texttt{goto } \ell_0\ \texttt{, inj}_1\ y_1^{T_1} \to \texttt{goto } \ell_1\ \texttt{\}}\end{array}}{\langle f;\ell \mid \mathbf{S} \mid \mathbf{F} + \{(x,a)\} \mid \mathbf{H}\rangle \to_{\mathcal{P}} \begin{array}{r}\langle \ell_i \mid \mathbf{S} \mid \mathbf{F} + \{(y_i,b)\} \mid [a, a+|T_0+T_1|) \triangleleft \mathbf{H} \\ +\mathcal{M}_{\mathbf{H}}(b, a+1, |T_i|)\rangle\end{array}}$$

$$\frac{\begin{array}{c} [c, c+|T_0|+|T_1|) \perp dom(\mathbf{H}) \\ \mathcal{P}_{f;\ell} = x^{T_0\times T_1}\texttt{:= (}y^{T_0}\texttt{, }z^{T_1}\texttt{); goto } \ell\texttt{'} \qquad A = [a_0, a_0+|T_0|) + [a_1, a_1+|T_1|)\end{array}}{\begin{array}{r}\langle f;\ell \mid \mathbf{S} \mid \mathbf{F} + \{(y,a_0),(z,a_1)\} \mid \mathbf{H}\rangle \to_{\mathcal{P}} \langle \ell' \mid \mathbf{S} \mid \mathbf{F} + \{(x,c)\} \\ \mid A \triangleleft \mathbf{H} + \mathcal{M}_{\mathbf{H}}(c, a_0, |T_0|) + \mathcal{M}_{\mathbf{H}}(c+|T_0|, a_1, |T_1|)\rangle\end{array}}$$

$$\frac{\begin{array}{c} A = [c, c+|T_0|+|T_1|) \\ \mathcal{P}_{f;\ell} = \texttt{let (}x^{T_0}\texttt{, }y^{T_1}\texttt{) := }z^{T_0\times T_1}\texttt{; goto } \ell\texttt{'} \qquad [a_0, a_0+|T_0|) \perp [a_1, a_1+|T_1|) \perp dom(\mathbf{H})\end{array}}{\begin{array}{r}\langle f;\ell \mid \mathbf{S} \mid \mathbf{F} + \{(z,c)\} \mid \mathbf{H}\rangle \to_{\mathcal{P}} \langle \ell' \mid \mathbf{S} \mid \mathbf{F} + \{(x,a_0),(y,a_1)\} \\ \mid A \triangleleft \mathbf{H} + \mathcal{M}_{\mathbf{H}}(a_0, c, |T_0|) + \mathcal{M}_{\mathbf{H}}(a_1, c+|T_0|, |T_1|)\rangle\end{array}}$$

$$\frac{\begin{array}{c} s = max(|T_0|, |T_1|) \\ \mathcal{P}_{f;\ell} = x^{T_0+T_1}\texttt{:= inj}_i\ y^T\texttt{; goto } \ell\texttt{'} \qquad A = [a, a+|T_i|) \qquad [b, b+s+1) \perp dom(\mathbf{H})\end{array}}{\langle f;\ell \mid \mathbf{S} \mid \mathbf{F} + \{(y,a)\} \mid \mathbf{H}\rangle \to_{\mathcal{P}} \langle \ell' \mid \mathbf{S} \mid \mathbf{F} + \{(x,b)\} \mid A \triangleleft \mathbf{H} + (b,i) + \mathcal{M}_{\mathbf{H}}(b+1, a, s)\rangle}$$

$$\frac{\mathcal{P}_{f;\ell} = \texttt{let x = g(x}_0\texttt{, .., x}_{n-1}\texttt{)}}{\langle f;\ell \mid \mathbf{S} \mid \mathbf{F} + \{(y_i,a_i) \mid i \in [n]\} \mid \mathbf{H}\rangle \to_{\mathcal{P}} \langle g \mid [\ell, x, \mathbf{F}]; \mathbf{S} \mid \{(x_i,a_i) \mid i \in [n]\} \mid \mathbf{H}\rangle}$$

$$\frac{\mathcal{P}_{f;\ell} = \texttt{return } x}{\langle f;\ell \mid [\ell', t, \mathbf{F}']; \mathbf{S} \mid \mathbf{F} \mid \mathbf{H}\rangle \to_{\mathcal{P}} \langle \ell' \mid \mathbf{S} \mid \mathbf{F}' + \{(y, \mathbf{F}(x))\} \mid \mathbf{H}\rangle}$$

C Translation from μMIR to μML

$$[\![\ell\texttt{: } x^{\texttt{box}\,T}\texttt{:= box } y^T\texttt{; goto } \ell\texttt{'}]\!] \triangleq \texttt{let } x\texttt{= } y \texttt{ in } \ell\ \vec{a}$$

$$[\![\ell\texttt{: } x^T\texttt{:= unbox } y^{\texttt{box}\,T}\texttt{; goto } \ell\texttt{'}]\!] \triangleq \texttt{let } x\texttt{= } y \texttt{ in } \ell\ \vec{a}$$

$$
\begin{array}{lcl}
[\![\ell\texttt{: } x^T \texttt{:= copy } y^T \texttt{; goto } \ell'\texttt{'}]\!] & \triangleq & \texttt{let } x\texttt{= } y \texttt{ in } \ell\ \vec{a} \\
[\![\ell\texttt{: } x^T \texttt{:= copy } *y^{PT} \texttt{; goto } \ell'\texttt{'}]\!] & \triangleq & \texttt{let } x\texttt{= } y \texttt{ in } \ell\ \vec{a} \\
[\![\ell\texttt{: } x^T \texttt{:= } y^{\texttt{int}} \texttt{op } z^{\texttt{int}} \texttt{; goto } \ell'\texttt{'}]\!] & \triangleq & \texttt{let x = y op z in } \ell\ \vec{a} \\
[\![\ell\texttt{: swap(} x^{PT}\texttt{, } y^{PT}\texttt{); goto } \ell'\texttt{'}]\!] & \triangleq & \texttt{let (}y\texttt{,}x\texttt{) = (}x\texttt{,}y\texttt{) in } \ell\ \vec{a} \\
[\![\ell\texttt{: drop } x^T \texttt{; goto } \ell'\texttt{'}]\!] & \triangleq & \texttt{();}\ \ell\ \vec{a} \\
[\![\ell\texttt{: assert } x^{bool} \texttt{; goto } \ell'\texttt{'}]\!] & \triangleq & \texttt{assert \{ } x \texttt{ \};}\ \ell\ \vec{a} \\
[\![\ell\texttt{: } x\texttt{:= C; goto } \ell'\texttt{'}]\!] & \triangleq & \texttt{let } x\texttt{= } C \texttt{ in } \ell\ \vec{a} \\
[\![\ell\texttt{: } x\texttt{:= g}\langle\texttt{..|..}\rangle\texttt{(}x_0\texttt{, ..,}x_{n-1}\texttt{); goto } \ell'\texttt{'}]\!] & \triangleq & \texttt{let } x \texttt{ =}f\texttt{(}x_0 \texttt{ , ..,}x\texttt{\_n) in } \ell\ \vec{a} \\
[\![\ell\texttt{: } x^{T_0\times T_1}\texttt{:= (}y^{T_0}\texttt{, } z^{T_1}\texttt{); goto } \ell'\texttt{'}]\!] & \triangleq & \texttt{let } x \texttt{ = (}y\texttt{,}z\texttt{) in } \ell\ \vec{a} \\
[\![\ell\texttt{: let (}x^{T_0}\texttt{, } y^{T_1}\texttt{) := } z^{T_0\times T_1}\texttt{; goto } \ell'\texttt{'}]\!] & \triangleq & \texttt{let (}x\texttt{,}y\texttt{) = } z \texttt{ in } \ell\ \vec{a} \\
[\![\ell\texttt{: } x^{T_0+T_1}\texttt{:= inj}_i\ y^T\texttt{; goto } \ell'\texttt{'}]\!] & \triangleq & \texttt{let } x \texttt{ = inj}_i\ z \texttt{ in } \ell\ \vec{a} \\
[\![\texttt{return } x]\!] & \triangleq & x \\
\left[\!\!\left[\begin{array}{l}\texttt{match } x^T \texttt{ \{} \\ \quad \texttt{inj}_0\ \ y_0{}^{T_0} \to \ell_0 \\ \quad \texttt{inj}_1\ \ y_1{}^{T_1} \to \ell_1 \\ \texttt{\}\}}\end{array}\right]\!\!\right] & \triangleq & \begin{array}{l}\texttt{begin match } x \texttt{ with} \\ \texttt{| inj}_0\ \ y_0\ \to\ \ell_0\ \vec{a} \\ \texttt{| inj}_1\ \ y_1\ \to\ \ell_1\ \vec{a} \\ \texttt{end}\end{array} \\
[\![\mathcal{P}]\!] & \triangleq & [\![\mathcal{P}_0]\!], ..., [\![\mathcal{P}_n]\!] \\
(\Delta, \ell, \texttt{fn } f\ \vec{a} \to T) & \triangleq & \texttt{rec fun } f\ \vec{a} = \ell\ \vec{a} \texttt{ and } [\![\Delta_0]\!] \texttt{ ... and } [\![\Delta_n]\!]
\end{array}
$$

D Proof of simulation preservation for μMIR

Definition D.1 (Simulation Invariant)**.** *The relation $\sim_{\mathcal{P}}$ between a well-typed μMIR configuration and a μML configuration is defined by the following conditions:*

$$
\begin{gathered}
\langle f; \ell \mid - \mid \mathbf{F} \mid \mathbf{H} \rangle \sim_{\mathcal{P}} \langle [\![\mathcal{P}_{f;\ell}, \mathbf{\Gamma}]\!] \mid E \mid K \rangle \\
K = ret\ E' \cdot ... \cdot ret\ E'' \\
\text{HeapEnv}(\mathbf{F}, \mathbf{H}, \mathbf{\Gamma}, E)
\end{gathered}
$$

Proof of Lemma 2.6. The proof proceeds by case-analysis on the transition relation $\to_{\mathcal{P}}$. Each case must shuffle memory around to show that the resulting configuration will readback in a manner which preserves the simulation.

Case (intro-pair) Recall that the reduction for introducing a pair is the following: $\langle f; \ell \mid \mathbf{S} \mid \mathbf{F} \oplus \{(y, a_0), (z, a_1)\} \mid \mathbf{H} \rangle \to_{\mathcal{P}} \langle \ell' \mid \mathbf{S} \mid \mathbf{F} \oplus \{(x, c)\} \mid \mathbf{H}' \rangle$

where

$$A = [a_0, a_0 + n_0) \oplus [a_1, a_1 + n_1)$$
$$\mathbf{H}' = A \triangleleft \mathbf{H} \oplus \mathcal{M}_{\mathbf{H}}(c, a_0, n_0) \oplus \mathcal{M}_{\mathbf{H}}(c \oplus n_0, a_1, n_1)$$

By inversion on $\sim_{\mathcal{P}}$, $K = \langle \texttt{let } x \texttt{ = (}y\texttt{,}z\texttt{) in } \ell \ \vec{a} \mid E \mid K \rangle$ which reduces to $K' = \langle \ell \ \vec{a} \mid E \oplus \{x, (y, z)\} \mid K \rangle$.

We know that y and z both have a readback judgement. When we translate their memories to the cells of x, that readback will remain the same. Then it's simple produce a readback for $\mathcal{R}(A \triangleleft (\mathbf{H}_1 \oplus \mathbf{H}_2) \oplus \mathcal{M}_{\mathbf{H}}(c, a_0, n_0) \oplus \mathcal{M}_{\mathbf{H}}(c + n_0, a_1, n_1), c, T_1 \times T_2, (v_1, v_2))$. With this we can translate the memory of C' to the environment of K', preserving the simulation.

Case (assert) When when we evaluate an assertion, $\texttt{assert } x^{bool}$, we know that $\mathbf{H}(\mathbf{F}(x)) = \texttt{true}$. By hypothesis, we know that x can be translated to a μMIR value, which tells us that $E(x) = \texttt{true}$. From this we can easily see that the assertion must then also evaluate to $\texttt{true}$ in μMIR. From this we preserve the simulation.

Case (swap) When we evaluate a swap, we produce from $C = \langle [f; \ell] \mid - \mid \mathbf{F} \mid \mathbf{H} \rangle$ a configuration who's heap is:

$$A = [\mathbf{H}(a), \mathbf{H}(a) + |T|) \cup [\mathbf{H}(b), \mathbf{H}(b) + |T|)$$
$$\mathbf{H}' = A \triangleleft \mathbf{H} + \mathcal{M}_{\mathbf{H}}(\mathbf{H}(a), \mathbf{H}(b), |T|) + \mathcal{M}_{\mathbf{H}}(\mathbf{H}(b), \mathbf{H}(a), |T|)$$

All this amounts to is that the memories of x and y are each translated to the other's position. The readbacks will therefore swap as well. This corresponds to the translation of the swap which also swaps the two variables in μML, preserving the simulation.

The other cases work in the same manner, memory is translated between cells following the motion of data in and out of sum and product types.

□

E Complete Operational Semantics of μML

$$\frac{}{\langle \texttt{ f e} \mid E \mid K \rangle \longrightarrow \langle \texttt{ f} \mid E \mid arg\ e \cdot K \rangle}$$

$$\frac{}{\langle \texttt{ v} \mid E \mid arg\ e \cdot K \rangle \longrightarrow \langle \texttt{ e} \mid E \mid fun\ v \cdot K \rangle}$$

$$\frac{}{\begin{array}{r}\langle \texttt{ v} \mid E \mid fun\ \texttt{rec f x = e and ... and g y = e'} \cdot K \rangle \longrightarrow \\ \langle\ e \mid [f \mapsto \texttt{rec f x = e and ... and g y = e'}, .., g \mapsto ...] \cdot E' \mid ret\ E \cdot K \rangle\end{array}}$$

$$\frac{}{\langle \texttt{ v} \mid E \mid ret\ E' \cdot K \rangle \longrightarrow \langle\ (\texttt{v}, E) \mid E' \mid K \rangle}$$

$$\frac{E(x) = (\texttt{rec f x = e and ... and g y = e'}, E')}{\langle \texttt{ v} \mid E \mid K \rangle \longrightarrow \langle\ fst\ E(x) \mid [x \mapsto fst\ E(x)] \cdot snd\ E(x) \mid K \rangle}$$

$$\frac{}{\langle\ \texttt{x} \mid E \mid K\ \rangle \longrightarrow \langle\ E(x) \mid E \mid K\ \rangle}$$

$$\frac{}{\langle\ \texttt{let x = e in e'} \mid E \mid K\ \rangle \longrightarrow \langle\ \texttt{e} \mid E \mid \mathit{let}\ x\ e' \cdot K\ \rangle}$$

$$\frac{}{\langle\ \texttt{v} \mid E \mid \mathit{let}\ x\ e' \cdot K\ \rangle \longrightarrow \langle\ \texttt{e'} \mid [x \mapsto (v, E)] \cdot E \mid K\ \rangle}$$

$$\frac{}{\langle\ \texttt{let (x, y) = e in e'} \mid E \mid K\ \rangle \longrightarrow \langle\ \texttt{e} \mid E \mid \mathit{unpair}\ x\ y\ e' \cdot K\ \rangle}$$

$$\frac{}{\langle\ \texttt{(}\mathtt{v}_0\ \texttt{,}\ \mathtt{v}_1\ \texttt{)} \mid E \mid \mathit{unpair}\ x\ y\ e' \cdot K\ \rangle \longrightarrow \langle\ \texttt{e'} \mid [x \mapsto (v_0, E)] \cdot [y \mapsto (v_1, E)] \cdot E' \mid K\ \rangle}$$

$$\frac{}{\begin{array}{r}\langle\ \texttt{match e with | inj}_0\ \texttt{x}_0\ \texttt{->e}_0\ \texttt{| inj}_1\ \texttt{x}_1\ \texttt{->e}_1\ \texttt{end} \mid E \mid K\ \rangle \longrightarrow \\ \langle\ \texttt{e} \mid E \mid \mathit{match}\ x_0\ e_0\ x_1\ e_1 \cdot K\ \rangle\end{array}}$$

$$\frac{}{\langle\ \texttt{inj}_i\ \texttt{v} \mid E \mid \mathit{match}\ x_0\ e_0\ x_1\ e_1 \cdot K\ \rangle \longrightarrow \langle\ \texttt{e}_i \mid [x_i \mapsto (v, E)] \cdot E' \mid K\ \rangle}$$

$$\frac{}{\langle\ \texttt{(e}_0\ \texttt{,}\ \texttt{e}_1\ \texttt{)} \mid E \mid K\ \rangle \longrightarrow \langle\ \texttt{e}_0 \mid E \mid \mathit{fst}\ e_1 \cdot K\ \rangle}$$

$$\frac{}{\langle\ \texttt{v}_0 \mid E \mid \mathit{fst}\ e_1 \cdot K\ \rangle \longrightarrow \langle\ \texttt{e}_1 \mid E \mid \mathit{snd}\ v_0 \cdot K\ \rangle}$$

$$\frac{x, y \notin \mathit{dom}(E)}{\langle\ \texttt{v}_1 \mid E \mid \mathit{snd}\ v_0 \cdot K\ \rangle \longrightarrow \langle\ \texttt{(x, y)} \mid [x \mapsto (v_1, E)] \cdot [y \mapsto (v_2, E')] \cdot E \mid K\ \rangle}$$

$$\frac{}{\langle\ \texttt{inj}_i\ \texttt{e} \mid E \mid K\ \rangle \longrightarrow \langle\ \texttt{e} \mid E \mid \mathit{inj}_i \cdot K\ \rangle}$$

$$\frac{}{\langle\ v \mid E \mid \mathit{inj}_i \cdot K\ \rangle \longrightarrow \langle\ \texttt{inj}_i\ \texttt{v} \mid E \mid K\ \rangle}$$

$$\frac{}{\langle\ \texttt{e}_0\ \texttt{op}\ \texttt{e}_1 \mid E \mid K\ \rangle \longrightarrow \langle\ e_0 \mid E \mid \mathit{leftop}\ op\ e_1 \cdot K\ \rangle}$$

$$\frac{}{\langle\ \texttt{v}_0 \mid E \mid \mathit{leftop}\ op\ e_1 \cdot K\ \rangle \longrightarrow \langle\ e_1 \mid E \mid \mathit{rightop}\ op\ v_0 \cdot K\ \rangle}$$

$$\frac{op((v_0, E), (v_1, E')) = r}{\langle\ \texttt{v}_1 \mid E \mid \mathit{rightop}\ op\ v_0 \cdot K\ \rangle \longrightarrow \langle\ r \mid E \mid K\ \rangle}$$

$$\frac{}{\langle\ \texttt{assert \{ e \}} \mid E \mid K\ \rangle \longrightarrow \langle\ e \mid E \mid \mathit{assert} \cdot K\ \rangle}$$

$$\frac{}{\langle\ \texttt{true} \mid E \mid \mathit{assert} \cdot K\ \rangle \longrightarrow \langle\ \texttt{()} \mid E \mid K\ \rangle}$$

$$\frac{}{\langle\ \texttt{e ; e'} \mid E \mid K\ \rangle \longrightarrow \langle\ \texttt{e} \mid E \mid \mathit{seq}\ e' \cdot K\ \rangle}$$

$$\frac{}{\langle\ \texttt{v} \mid E \mid \mathit{seq}\ e' \cdot K\ \rangle \longrightarrow \langle\ \texttt{e'} \mid E \mid K\ \rangle}$$

F Complete Type System of MiniMir

$$\frac{y : T \in \mathbf{\Gamma} \qquad \mathit{Copy}(T)}{\mathbf{\Sigma}; \mathbf{\Gamma}; \mathbf{L} \vdash_f x^T \texttt{:= copy } y^T \dashv \mathbf{\Gamma}, x : T; \mathbf{L}} \text{ (COPY-VAL)}$$

$$\frac{y : P\ T \in \mathbf{\Gamma} \qquad P \in \{\texttt{\&mut}_\alpha, \texttt{\&}_\alpha, \texttt{box}\} \qquad \mathit{Copy}(T)}{\mathbf{\Sigma}; \mathbf{\Gamma}; \mathbf{L} \vdash_f x^T \texttt{:= copy } *y^{P\ T} \dashv \mathbf{\Gamma}, x : T; \mathbf{L}} \text{ (COPY-REF)}$$

$$\frac{}{\mathbf{\Sigma}; y : \texttt{box } T, \mathbf{\Gamma}; \mathbf{L} \vdash_f x^T \texttt{:= unbox } y^{\texttt{box } T} \dashv \mathbf{\Gamma}, x : T; \mathbf{L}} \text{ (UNBOX)}$$

$$\frac{}{\mathbf{\Sigma}; y : T, \mathbf{\Gamma}; \mathbf{L} \vdash_f x^{\texttt{box } T} \texttt{:= box } y^T \dashv \mathbf{\Gamma}, x : \texttt{box } T; \mathbf{L}} \text{ (BOX)}$$

$$\frac{}{\mathbf{\Sigma}; y : T, \mathbf{\Gamma}; \mathbf{L} \vdash_f x^{\texttt{\&mut}_\alpha\ T} \texttt{:= \&mut}_\alpha y^T \dashv \mathbf{\Gamma}, x : \texttt{\&mut}_\alpha\ T, y :^{\dagger\alpha} T; \mathbf{L}} \text{ (BORROW-MUT)}$$

$$\frac{P \in \{\texttt{box}, \texttt{\&mut}_\beta\} \qquad \alpha \leq \beta \in \mathbf{L}}{\mathbf{\Sigma}; y : \texttt{\&mut}_\alpha\ P\ T, \mathbf{\Gamma}; \mathbf{L} \vdash_f x^{\texttt{\&mut}_\alpha\ T} \texttt{:= unnest } y^{\texttt{\&mut}_\alpha P\ T} \dashv \mathbf{\Gamma}, x : \texttt{\&mut}_\alpha\ T; \mathbf{L}} \text{ (UNNEST)}$$

$$\frac{P \in \{\texttt{\&mut}_\alpha, \texttt{box}\} \qquad x : P\,T, y : P\,T \in \mathbf{\Gamma}}{\mathbf{\Sigma}; \mathbf{\Gamma}; \mathbf{L} \vdash_f \texttt{swap}(x^{P\ T}\texttt{, } y^{P\ T}) \dashv \mathbf{\Gamma}; \mathbf{L}} \text{ (SWAP)} \qquad \frac{T \in \{\texttt{int}, \texttt{unit}, \texttt{\&}_\alpha\}}{\mathbf{\Sigma}; x : T, \mathbf{\Gamma}; \mathbf{L} \vdash_f \texttt{drop } x^T \dashv \mathbf{\Gamma}; \mathbf{L}} \text{ (DROP)}$$

$$\frac{}{\mathbf{\Sigma}; x : \texttt{\&mut}_\alpha\ T, \mathbf{\Gamma}; \mathbf{L} \vdash_f \texttt{immut x} \dashv \mathbf{\Gamma}, x : \texttt{\&}_\alpha\ T; \mathbf{L}} \text{ (IMMUT)}$$

$$\frac{}{\mathbf{\Sigma}; y : T_i, \mathbf{\Gamma}; \mathbf{L} \vdash_f x^{T_0+T_1} \texttt{:= inj}_i\ y^T \dashv \mathbf{\Gamma}, x : T_1 + T_2; \mathbf{L}} \text{ (INTRO-SUM)}$$

$$\frac{}{\mathbf{\Sigma}; y : T_1, z : T_2, \mathbf{\Gamma}; \mathbf{L} \vdash_f x^{T_0 \times T_1} \texttt{:= (}y^{T_0}\texttt{, } z^{T_1}\texttt{)} \dashv \mathbf{\Gamma}, x : T_1 \times T_2; \mathbf{L}} \text{ (INTRO-PAIR)}$$

$$\frac{}{\mathbf{\Sigma}; \mathbf{\Gamma}; \mathbf{L} \vdash_f \texttt{let x = C} \dashv \mathbf{\Gamma}, x : T_C; \mathbf{L}} \text{ (CONST-INTRO)} \qquad \frac{x : \texttt{bool} \in \mathbf{\Gamma}}{\mathbf{\Sigma}; \mathbf{\Gamma}; \mathbf{L} \vdash_f \texttt{assert x} \dashv \mathbf{\Gamma}; \mathbf{L}} \text{ (ASSERT)}$$

$$\frac{y : \texttt{int}, z : \texttt{int} \in \mathbf{\Gamma}}{\mathbf{\Sigma}; \mathbf{\Gamma}; \mathbf{L} \vdash_f x^T \texttt{:= } y^{\texttt{int}} \texttt{op } z^{\texttt{int}} \dashv \mathbf{\Gamma}, x : T_{op}; \mathbf{L}} \text{ (OP)}$$

$$\frac{}{\mathbf{\Sigma}; z : T_1 \times T_2, \mathbf{\Gamma}; \mathbf{L} \vdash_f \texttt{let (}x^{T_0}\texttt{, } y^{T_1}\texttt{) := } z^{T_0 \times T_1} \dashv \mathbf{\Gamma}, x : T_1, y : T_2; \mathbf{L}} \text{ (ELIM-PAIR)}$$

$$\frac{P \in \{\texttt{\&mut}_\alpha, \texttt{\&}_\alpha\}}{\mathbf{\Sigma}; z : P\ (T_1 \times T_2), \mathbf{\Gamma}; \mathbf{L} \vdash_f \texttt{let (ref x, ref y) = z} \dashv \mathbf{\Gamma}, x : P\ T_1, y : P\ T_2; \mathbf{L}} \text{ (ELIM-PAIR-REF)}$$

$$\frac{\begin{array}{c}\forall j, 0 \leq j < l, \beta_{a_j} \leq \beta_{b_j} \in \mathbf{L} \quad \forall i, 0 \leq i < n+1, T_i = T'_i[\alpha_0/\beta_0, .., \alpha_{m-1}/\beta_{m-1}] \\ \texttt{fn g}\langle \alpha_0, ..., \alpha_{m-1} \mid \alpha_{a_0} \leq \alpha_{b_0}, ..., \alpha_{a_{l-1}} \leq \alpha_{b_{l-1}} \rangle (x_0 : T'_0, ...x_{n-1} : T'_{n-1}) \to T'_n \in \mathbf{\Sigma}\end{array}}{\mathbf{\Sigma}; x_0 : T_0, .., x_{n-1} : T_{n-1}, \mathbf{\Gamma}; \mathbf{L} \vdash_f \texttt{let x = g}\langle \beta_0, .., \beta_{m-1} \rangle \texttt{(x}_0\texttt{, .., x}_{n-1}\texttt{)} \dashv \mathbf{\Gamma}, x : T_n; \mathbf{L}} \text{ (CALL)}$$

$$\frac{}{\mathbf{\Sigma}; \mathbf{\Gamma}; \mathbf{L} \vdash_f \alpha \leq \beta \dashv \mathbf{\Gamma}; \mathbf{L}, \alpha \leq \beta, \mathbf{L}} \text{ (SUB-LIFETIME)}$$

$$\frac{\begin{array}{ccc} & \mathbf{L}' = \mathbf{L} \setminus \{\alpha \leq \beta \mid \forall \beta, \alpha \leq \beta \in \mathbf{L}\} & \nexists \beta, \beta \leq \alpha \in \mathbf{L} \\ \alpha \notin \mathbf{\Sigma}_{\text{exp}f} & \mathbf{\Gamma}' = \{thaw_\alpha(v) \mid v \in \mathbf{\Gamma}\} & \forall\, x :^{\mathbf{n}} T \in \mathbf{\Gamma}', \alpha \notin \text{lifetimes}(T) \end{array}}{\mathbf{\Sigma}; \mathbf{\Gamma}; \mathbf{L} \vdash_f \texttt{thaw } \alpha \dashv \mathbf{\Gamma}'; \mathbf{L}'} \text{ (THAW)}$$

where $\mathbf{\Sigma}_{exp\ f}$ is the set of lifetimes that survive the function f.

$$thaw_\alpha(v) = \begin{cases} x : T & v = x :^{\dagger\alpha} T \\ v & otherwise \end{cases}$$

$$\frac{\mathbf{\Sigma}; \mathbf{\Xi}_a; \mathbf{\Lambda}_a \vdash_f I \dashv \mathbf{\Sigma}; \mathbf{\Xi}_\ell; \mathbf{\Lambda}_\ell}{\mathbf{\Sigma}; \mathbf{\Xi}; \mathbf{\Lambda} \vdash_{f,a} \texttt{I; goto } \ell} \text{ (SEQUENCE)} \qquad \frac{\mathbf{\Xi}_a = \{x : \mathbf{\Sigma}_{ret f}\}}{\mathbf{\Sigma}; \mathbf{\Xi}; \mathbf{\Lambda} \vdash_{f,a} \texttt{return x}} \text{ (RETURN)}$$

where $\mathbf{\Sigma}_{ret\ f} = \{T_n \mid \texttt{fn g } \langle ... \mid ... \rangle (x_0 : T'_0, ...x_{n-1} : T'_{n-1}) \to T'_n = \mathbf{\Sigma}(f)\}$

$$\frac{\mathbf{\Xi}_{\ell_i} = y_i : T_i, \mathbf{\Xi}_a \quad \mathbf{\Lambda}_{\ell_i} = \mathbf{\Lambda}_a \quad i \in \{0, 1\}}{\mathbf{\Sigma}; x : T_0 + T_1, \mathbf{\Xi}_a; \mathbf{\Lambda}_a \vdash_{f,a} \begin{array}{l}\texttt{match } x^{T_0+T_1}\texttt{\{} \\ \quad \texttt{inj}_0\ y_0^{T_0} \to \ell_0 \\ \quad \texttt{inj}_1\ y_1^{T_1} \to \ell_1 \\ \texttt{\}}\end{array}} \text{ (MATCH-VAL)}$$

$$\frac{\mathbf{\Xi}_i = y_i : P\ T_i, \mathbf{\Xi}_a \quad \mathbf{\Lambda}_i = \mathbf{\Lambda}_a \quad i \in \{0, 1\} \quad P \in \{\texttt{\&mut}_\alpha, \texttt{\&}_\alpha\}}{\mathbf{\Sigma}; x : P\ (T_0 + T_1), \mathbf{\Xi}_a; \mathbf{\Lambda}_a \vdash_{f,a} \begin{array}{l}\texttt{match } *x^{P\ T_0+T_1}\texttt{\{} \\ \quad \texttt{inj}_0\ \ \texttt{ref}\ \ y_0^{P\ T_0} \to \ell_0 \\ \quad \texttt{inj}_1\ \ \texttt{ref}\ \ y_1^{P\ T_1} \to \ell_1 \\ \texttt{\}}\end{array}} \text{ (MATCH-REF)}$$

$$\frac{\begin{array}{c}\mathbf{\Xi}_\ell = \bigcup x_i : T_i \quad \mathbf{\Lambda}_\ell = \{\beta_{a_j} \leq \beta_{b_j} \mid \forall j, 0 \leq j < l\} \quad \forall S \in \Delta, \mathbf{\Sigma}; \mathbf{\Xi} \vdash_{f,a} S \\ \texttt{fn f}\langle \alpha_0, .., \alpha_{m-1} \mid \alpha_{a_0} \leq \alpha_{b_0}, .., \alpha_{a_{l-1}} \leq \alpha_{b_{l-1}} \rangle (x_0 : T'_0, .., x_{n-1} : T'_{n-1}) \to T'_n\end{array}}{\mathbf{\Sigma}, \mathbf{\Xi} \vdash (\Delta, \ell, \sigma)} \text{ (FUNCTION)}$$

$$\frac{\mathbf{\Sigma} = \{\sigma \mid \forall (\Delta, \ell, \sigma) \in \mathcal{P}\} \quad \forall f \in \mathcal{P}, \mathbf{\Sigma} \vdash f}{\vdash \mathcal{P}} \text{ (PROGRAM)}$$

G Complete Operational Semantics of MiniMir

The semantics of μMIR are extended with the following rules:

$$\frac{\mathcal{P}_{f;\ell} = \texttt{immut } y^{\texttt{\&mut}_\alpha\, T}\texttt{; goto } \ell\texttt{'}}{\langle \ell \mid \mathbf{S} \mid \mathbf{F} \mid \mathbf{H}\rangle \to_{\mathcal{P}} \langle \ell' \mid \mathbf{S} \mid \mathbf{F} \mid \mathbf{H}\rangle} \text{ (IMMUT)}$$

$$\frac{\mathcal{P}_{f;\ell} = x^{\texttt{\&mut}_\alpha\, T}\texttt{:= \&mut}_\alpha y^{T}\texttt{; goto } \ell\texttt{'} \qquad \mathbf{F}(y) = a \qquad b \notin dom(\mathbf{H})}{\langle \ell \mid \mathbf{S} \mid \mathbf{F} \mid \mathbf{H}\rangle \to_{\mathcal{P}} \langle \ell' \mid \mathbf{S} \mid \mathbf{F} + \{(x,b)\} \mid \mathbf{H} + \{(b,a)\}\rangle} \text{ (BORROW-MUT)}$$

$$\frac{\mathcal{P}_{f;\ell} = x^{\texttt{\&mut}_\alpha\, T}\texttt{:= unnest } y^{\texttt{\&mut}_\alpha P\, T}\texttt{; goto } \ell\texttt{'} \qquad b \notin dom(\mathbf{H})}{\langle \ell \mid \mathbf{S} \mid \mathbf{F} + \{(y,a)\} \mid \mathbf{H} + \{(a,p)\}\rangle \to_{\mathcal{P}} \langle \ell' \mid \mathbf{S} \mid \mathbf{F} + \{(x,b)\} \mid \mathbf{H} + \{(b,\mathbf{H}(p))\}\rangle} \text{ (UNNEST)}$$

$$\frac{\mathcal{P}_{f;\ell} = \texttt{match *x \{ .. \}} \qquad i = H(a)}{\langle \ell \mid \mathbf{S} \mid \mathbf{F} + \{(x,a)\} \mid \mathbf{H}\rangle \to_{\mathcal{P}} \langle \ell_i \mid \mathbf{S} \mid \mathbf{F} + \{(y_i, \mathbf{H}(a)+1)\} \mid \mathbf{H}\rangle} \text{ (MATCH-REF)}$$

H Translation from MiniMir to μML

$[\![\ell\texttt{: } x^{T}\texttt{:= copy *}y^{P\,T}\texttt{; goto } \ell\texttt{'}]\!]$ $\triangleq$ `let x = *y in` $\ell\ \vec{a}$ $P \in \{\texttt{\&}_\alpha, \texttt{\&mut}_\alpha\}$

$[\![\ell\texttt{: } x^{\texttt{\&mut}_\alpha\,T}\texttt{:= \&mut}_\alpha y^{T}\texttt{; goto } \ell\texttt{'}]\!]$ $\triangleq$
```
let x = (y, any) in
let y = ^x in
ℓ a⃗
```

$[\![\ell\texttt{: } x^{\texttt{\&mut}_\alpha\,T}\texttt{:= unnest } y^{\texttt{\&mut}_\alpha\,P\,T}\texttt{; goto } \ell\texttt{'}]\!]$ $\triangleq$
```
let x = (**y, *^y) in
assume { ^*y = ^^y };
ℓ a⃗
```
$P = \texttt{\&mut}_\beta$

$[\![\ell\texttt{: } x^{\texttt{\&mut}_\alpha\,T}\texttt{:= unnest } y^{\texttt{\&mut}_\alpha\,P\,T}\texttt{; goto } \ell\texttt{'}]\!]$ $\triangleq$ `let x = (*y, ^y ) in` $\ell\ \vec{a}$ $P = \texttt{box}$

$[\![\ell\texttt{: immut } y^{\texttt{\&mut}_\alpha\,T}\texttt{; goto } \ell\texttt{'}]\!]$ $\triangleq$ `assume { *y = ^y};` $\ell\ \vec{a}$

$[\![\ell\texttt{: swap(}x^{P\,T}\texttt{, } y^{P\,T}\texttt{); goto } \ell\texttt{'}]\!]$ $\triangleq$
```
let t = *x in
let x = (*y, ^x) in
let y = (t, ^y) in
ℓ a⃗
```
P $= \texttt{\&mut}_\alpha$T

$\triangleq$ `let (y,x) = (x,y) in` $\ell\ \vec{a}$ otherwise

$[\![\ell\texttt{: thaw } \alpha\texttt{; goto } \ell\texttt{'}]\!]$ $\triangleq$ `()`

$[\![\ell\texttt{: } \alpha \leq \beta\texttt{; goto } \ell\texttt{'}]\!]$ $\triangleq$ `()`

$[\![\ell\texttt{: let (ref } x^{P\,T_0}\texttt{, ref } y^{P\,T_1}\texttt{) := *}z^{P\,(T_0\times T_1)}\texttt{; goto } \ell\texttt{'}]\!]$ $\triangleq$
```
let (x_n, y_n) = *z in
let (x_e, y_e) = ^z in
let x = (x_n, x_e) in
let y = (y_n, y_e) in
ℓ a⃗
```
$P = \texttt{\&mut}_\alpha$

$[\![\ell\texttt{: let (ref } x^{P\,T_0}\texttt{, ref } y^{P\,T_1}\texttt{) := *}z^{P\,(T_0\times T_1)}\texttt{; goto } \ell\texttt{'}]\!]$ $\triangleq$ `let (x, y) = z` $P = \texttt{\&}_\alpha$

$$\left[\!\left[\begin{array}{l}\texttt{match *x}^{T}\ \texttt{\{}\\ \quad \texttt{inj}_0\ \ \texttt{ref y}_0 \overset{P\,T_0}{\hookrightarrow} \ell_0\\ \quad \texttt{inj}_1\ \ \texttt{ref y}_1 \overset{P\,T_1}{\hookrightarrow} \ell_1\\ \texttt{\}}\end{array}\right]\!\right]$$

$\triangleq$

```
begin match *x with
| inj0  y0 →
let y0 = (y0, any) in
assume { ^x = inj0  ^y0 };
let x = (inj0  ^y0, ^x) in
L0
| inj1  y1 →
let y1 = (y1, any);
assume { ^x = inj1  ^y1 };
let x = (inj1  ^y1, ^x) in
L1
end
```

I Proof of simulation preservation for MiniMir

Definition I.1 (Heap Fragment Type for MiniMir)**.** *The heap types of MiniMir extend the judgements of μMIR with a borrow store, checking that every borrow has a source and that they agree on typing.*

$$\frac{\mathcal{B}_1; \mathbf{H}_1 \models a :^{\mathbf{n}} T_1 \qquad \mathbf{H} = \mathbf{H}_1 + \mathbf{H}_2}{\mathcal{B}_2; \mathbf{H}_2 \models a + |T_1| :^{\mathbf{n}} T_2 \qquad \mathcal{B} = \mathcal{B}_1 + \mathcal{B}_2}$$
$$\mathcal{B}; \mathbf{H} \models a :^{\mathbf{n}} T_1 \times T_2$$

$$\frac{T \in \{\texttt{int}, \texttt{bool}, \texttt{unit}\} \qquad c \textit{ is a constant of type } T}{\varnothing; \{(a, c)\} \models a :^{\mathbf{n}} T}$$

$$\frac{\mathcal{B}; \mathbf{H} \models a + 1 :^{\mathbf{n}} T_i}{\mathcal{B}; \mathbf{H} \oplus \{(a, i)\} \models a :^{\mathbf{n}} T_0 + T_1}$$

$$\frac{}{\mathbf{give}^m_\alpha(a, T); \varnothing \models a :^{\dagger\alpha} T}$$

$$\frac{\mathcal{B}; \mathbf{H} \models a :^{\dagger\alpha} T}{\mathbf{give}^i_\alpha(a, T), \mathcal{B}; \mathbf{H} \models a :^{\dagger\alpha} T}$$

$$\frac{\mathcal{B}; \mathbf{H} \models a :^{\mathbf{n}} T}{\mathbf{take}^m_\alpha(a, T), \mathcal{B}; \{(p, a)\} \oplus \mathbf{H} \models p :^{\mathbf{n}} \texttt{\&mut}_\alpha T}$$

$$\frac{}{\mathbf{take}^i_\alpha(a, T); \{(p, a)\} \models p :^{\mathbf{n}} \texttt{\&}_\alpha T}$$

Theorem I.2 (Type Preservation)**.** *Given a well-typed MiniMir configuration C, if $C \to_{\mathcal{P}} C'$, then C' is well-typed.*

Proof. The proof proceeds by case-analysis on the reductions of MiniMir. We will show several cases which illustrate preservation for owned data, mutable borrows and immutable borrows.

Case (intro-pair) Given a configuration C with code $x^{T_0 \times T_1} \texttt{:= (}y^{T_0}\texttt{, }z^{T_1}\texttt{)}$, we know that that we must have $\mathcal{B}_y; \mathbf{H}_y \models a_0 :^{\cdot} T_1$, and $\mathcal{B}_z; \mathbf{H}_z \models a_1 :^{\cdot} T_2$. We from this we construct $\mathcal{B}_y + \mathcal{B}_z; \mathbf{H}_x \models c :^{\cdot} T_1 \times T_2$, here $dom(\mathbf{H}_x) = [c, c + |T_0 \times T_1|)$. It's immediately apparent that when we move the memories of y and z into the cells of $\mathbf{H}_x$, we obtain the desired heap typing.

Case (borrow-mut) The reduction of C with code $x^{\texttt{\&mut}_\alpha T} \texttt{:= \&mut}_\alpha y^T$, adds a new pointer x holding the address of y to $\mathbf{H}$. By hypothesis we have $\mathcal{B}_y; \mathbf{H}_y \models a :^{\cdot} T$. We need to produce two new judgements after evaluating this instruction, one for $x : \texttt{\&mut}_\alpha T$ and one for $y :^{\dagger\alpha} T$. To preserve typing we add a pair of tokens $\mathbf{give}^r_\alpha(a, T), \mathbf{take}^r_\alpha(a, T)$ to $\mathcal{B}$. Using these tokens we can construct $\{\mathbf{give}^r_\alpha(a, T)\}; \varnothing \models a :^{\dagger\alpha} T$ for y and $\{\mathbf{take}^r_\alpha(a, T)\} \cup \mathcal{B}_y; \{(p, a)\} \oplus \mathbf{H}_y \models a :^{\cdot} \texttt{\&mut}_\alpha T$.

Case (immut) By hypothesis we have $\{\mathbf{take}^r_\alpha(a, T)\} \cup \mathcal{B}_x; \{(p, a)\} \oplus \mathbf{H}_x \models p :^{\cdot} \texttt{\&mut}_\alpha T$. By the safety of $\mathcal{B}$ we know there must be $\{\mathbf{give}^r_\alpha(a, T)\} \cup \mathcal{B}_y; \mathbf{H}_y \models a' :^{\dagger\alpha} T'$. By inversion, this judgement must contain the a subtree for $\{\mathbf{give}^r_\alpha(a, T)\}; \varnothing \models a :^{\dagger\alpha} T$. We must transform the judgement for x into one for an immutable reference. We do this by removing $\mathbf{give}^r_\alpha(a, T), \mathbf{take}^r_\alpha(a, T)$ and inserting $\mathbf{give}^i_\alpha(a, T), \mathbf{take}^i_\alpha(a, T)$ into the borrow store. As a result the ownership of $\mathbf{H}_x$

□

Here we give the complete definition of the readback for MiniMir.

Definition I.3 (Readback)**.** *The readback of a MiniMir heap constructs a map of address and type to MiniML value.*

$$\frac{\mathcal{R}^{\mathbf{n}}(\mathcal{B}_1, A, \mathbf{H}_1, a, T_1, E_1) \qquad \mathbf{H} = \mathbf{H}_1 + \mathbf{H}_2}{\mathcal{R}^{\mathbf{n}}(\mathcal{B}_2, A, \mathbf{H}_2, a + |T_1|, T_2, E_2) \qquad \mathcal{B} = \mathcal{B}_1 + \mathcal{B}_2}$$
$$\overline{\mathcal{R}^{\mathbf{n}}(\mathcal{B}, A, \mathbf{H}, a, T_1 \times T_2, E_1 + E_2 + (a, (E_1(a), E_2(a + |T_1|))))}$$

$$\frac{\mathcal{R}^{\mathbf{n}}(\mathcal{B}, A, \mathbf{H}, a + 1, T_i, E_i)}{\mathcal{R}^{\mathbf{n}}(\mathcal{B}, A, \{(a, i)\} \oplus \mathbf{H}, a, T_1 + T_2, E_i(a, \mathit{inj}_i E_i(a)))}$$

$$\frac{T \in \{\texttt{int}, \texttt{bool}, \texttt{unit}\} \qquad c \text{ is a constant of type } T}{\mathcal{R}^{\mathbf{n}}(\varnothing, A, \{(a, c)\}, a, T, (a, c))}$$

$$\frac{A(a, T) = v}{\mathcal{R}^{\dagger\alpha}(\{\mathbf{give}^m_\alpha(a, T)\}, A, \varnothing, a, T, v)} \qquad \frac{\mathcal{R}^{\dagger\alpha}(\mathcal{B}, A, \mathbf{H}, a, T, v)}{\mathcal{R}^{\dagger\alpha}(\{\mathbf{give}^i_\alpha(a, T)\} \cup \mathcal{B}, A, \mathbf{H}, a, T, v)}$$
$$\frac{A(a, T) = v}{\mathcal{R}^{\mathbf{n}}(\{\mathbf{take}^i_\alpha(a, T)\}, A, \{(p, a)\}, p, \texttt{\&}_\alpha\ T, v)}$$

$$\frac{\mathcal{R}^{\mathbf{n}}(\mathcal{B}, A, \{a\} \triangleleft \mathbf{H}, \mathbf{H}(a), T, E) \qquad A(\mathbf{H}(a), T) = v}{\mathcal{R}^{\mathbf{n}}(\mathbf{take}^m_\alpha(a, T) \cup \mathcal{B}, A, \{(p, a)\} \oplus \mathbf{H}, p, \texttt{\&mut}_\alpha\ T, E + (a, (E(\mathbf{H}(a)), v)))}$$

Lemma I.4 (Progress)**.** *Given a MiniMir trace $C \to^*_{\mathcal{P}} C'$ and a MiniML configuration K such that $C \to^*_{\mathcal{P}} C' \sim_{\mathcal{P}} K$, if K is not stuck then C is not stuck.*

Lemma I.4. The proof proceeds by case analysis on the reductions of MiniML. In each step we exploit our hypothesis $\text{HeapEnv}(\mathbf{F}, \mathbf{H}, \mathbf{\Gamma}, E)$ to show that the MiniMir heap must be well-formed and allow reduction. In the case of assertions we note that for the MiniML assertion to evaluate to `true`, the readback requires the MiniMir heap to also hold `true`. □

Lemma I.5 (Terminal Configurations)**.** *Given a terminal MiniML configuration K, for any MiniMir trace Θ such that $\Theta \sim_{\mathcal{P}} K$, then $\Theta = C \not\to_{\mathcal{P}}$ and C terminal.*

Proof. The only terminal configurations for MiniML possible with the simulation are those in relation with `return` instructions inside the main function of the MiniMir program. □

Lemma I.6. *If $\mathcal{R}^{\dagger\alpha}(\mathcal{B}_x, A, \mathbf{H}_x, \mathbf{F}(x), T, V_x)$, then for all $\mathcal{R}^{\bullet}(\mathcal{B}'_x, A', \mathbf{H}'_x, \mathbf{F}(x), T, V'_x)$, $dom(V_x) \subseteq dom(V'_x)$.*

Proof. By induction over the type T we observe that when we perform a readback with a frozen activeness $\dagger\alpha$ we will leave out subtrees of the active readback. □

Definition I.7 (McFly)**.** *The* McFly *relation calculates the required prophecies from a MiniMir trace.*

$$\frac{\text{HeapEnv}\ (A, \mathcal{B}, \mathbf{F}', \mathbf{H}', \mathbf{\Xi}_{f;\ell}, E)}{\mathcal{P}_{f;l} = \texttt{thaw}\ \alpha \qquad A' = A \oplus \bigoplus_{\mathbf{give}^r_\alpha(a,T) \in \mathcal{B}} \{E \mid \mathcal{R}^{\dagger\alpha}(\mathcal{B}'_a, A, \mathbf{H}'_a, a, \mathbf{\Xi}_{f';\ell'}(a, \bullet), E)\}}$$
$$\overline{\text{McFly}^*(A', \langle [f; \ell] \mid - \mid \mathbf{F} \mid \mathbf{H} \rangle \to_{\mathcal{P}} \langle f'; \ell' \mid - \mid \mathbf{F}' \mid \mathbf{H}' \mid \mathcal{B}' \rangle, A)}$$

At the end of a trace we may still have active borrows (if the trace is stuck), to handle this situation, we perform a readback of each borrowed variable. During this readback we do not require the heap to be separated, which avoids the need for any prophecies by allowing cells to be reused in several readbacks.

$$\frac{\text{McFly}^*(A, \mathcal{C} \to^*_{\mathcal{P}} C', E) \qquad E = \bigoplus E_a \qquad \forall \mathbf{give}^r_\alpha(a, T) \in \mathcal{B}, \exists \mathbf{H}_a \subseteq \mathbf{H}, \mathcal{B}_a \subseteq \mathcal{B}, \mathcal{R}^{\cdot}(\mathcal{B}_a, \varnothing, \mathbf{H}_a, a, T, E_a)}{\text{McFly}(A, C \to^*_{\mathcal{P}} \langle f; \ell' \mid - \mid \mathbf{F} \mid \mathbf{H}\rangle)}$$

Lemma I.8. *Given a trace of well-typed MiniMir configurations* Θ *and a prophecy map* A *such that* McFly(A, Θ)*, then* HeapEnv $(A, \mathcal{B}, \mathbf{F}, \mathbf{H}, \Xi_{\Theta_0}, E)$.

Proof. We find ourselves required to prove that for all variables in the frame of the initial configuration Θ_0 we have a readback. We prove this by induction on the type of the variable. Whenever we encounter a borrow, we must show that there is a prophecy in A for this borrow. By inversion on McFly, either there is a thaw for the lifetime of the borrow and we have a prophecy, or it is active until the end of the trace and we must also have a prophecy. Using this we can construct the readback for every variable and thus translate the heap to an environment. □

Definition I.9 (Simulation Relation)**.** *The relation* $\sim_{\mathcal{P}}$ *between a well-typed MiniMir trace* Θ *and a Krivine configuration is defined by the following conditions:*

$$\begin{aligned}\Theta = \langle f; \ell \mid \mathbf{S} \mid \mathbf{F} \mid \mathbf{H} \mid \mathcal{B}\rangle \to^*_{\mathcal{P}} C \sim_{\mathcal{P}} \langle [\![\mathcal{P}_l]\!] \mid E \mid K\rangle \\ \text{HeapEnv } (A, \mathcal{B}, \mathbf{F}, \mathbf{H}, \Xi_\ell, E) \\ K = ret\ E' \cdot \ldots \cdot ret\ E'' \\ \text{McFly}(A, \langle f; \ell \mid \mathbf{S} \mid \mathbf{F} \mid \mathbf{H} \mid \mathcal{B}\rangle \to^*_{\mathcal{P}} C)\end{aligned}$$

Lemma 3.8. The proof proceeds by case-analysis on the reduction $C \to_{\mathcal{P}} C'$. The cases related to owned data ((INTRO-PAIR), (CONSTANT)) proceed like in μMIR. The most interesting cases of this proof are those for creating and freezing a borrow.

Case (borrow-mut) To preserve our simulation, we must ensure that the translated code reduces properly to the next label and that the resulting memories stay linked by HeapEnv. Because we create a new borrow, to have a readback, we need a prophecy map which includes this new borrow. We get this by Lemma I.8, we know that there must be a prophecy for the borrowed variable.

Case (immut) As a reminder, the operational semantics tell us that the operation is a no-op. The translation of `immut` gives us

$$K = \langle \texttt{assume \{ * x = \^{} x \}; } \ell'\ \vec{a} \mid E \mid K\rangle$$

To preserve the simulation, the machine must reduce the `assume`, which means proving that `*x = ^x`.

Let $\mathbf{F}(x) = a$, by inversion on the HeapEnv, we know that there is $\mathbf{H}_x, V_x, \mathcal{B}_x$ such that $\mathcal{R}^\bullet(\mathbf{take}^m_\alpha(a, T) \cup \mathcal{B}_x, A, \mathbf{H}_x, a, \texttt{\&mut}_\alpha T_x, V_x)$ so, $E(x) = (V_x(\mathbf{H}(a), T_x), A(\mathbf{H}(a), T_x))$. By the safety of $\mathcal{B}$, there must be a variable y which consumes $\mathbf{give}^m_\alpha(a, T)$, by assigning y the memory $\mathbf{H}_x$ in C', we can preserve the existence of the readback.

Let $\mathcal{R}^{\dagger\alpha}(\mathcal{B}_y, \mathbf{H}_x, \mathbf{F}(y), T, V_y,)$ be the readback of y in C'. We know that $(\mathbf{H}(a), T_x) \in dom(V_y)$.

Let C'' be either the first configuration in Θ such that $\mathcal{P}_{f;\ell''} = \texttt{thaw}\ \alpha$ or the last configuration of Θ. By applying Lemma I.8, we know that C'' must have a readback. Since $(\mathbf{H}(a), T_x) \in dom(V_y)$, by Lemma I.6, $(\mathbf{H}(a), T_x)$ must be included in the readback of y at C''. This means that by inversion on the readback of C'', there must be a $\mathcal{R}^{\dagger\alpha}(\mathcal{B}'_x, A', \mathbf{H}'_x, \mathbf{H}(a), T_x, V'_x)$.

Because we froze all the memory associated with $\mathbf{H}(a)$ at C, we know that $\mathcal{B}'_x \subseteq \mathcal{B}_x$ and $\mathbf{H}_x \subseteq \mathbf{H}'_x$. By typing we know that $\mathcal{B}_x$ can only have immutable give and take tokens. Since that means we control all the memory, $\mathbf{H}_x = \mathbf{H}'_x$. Finally since $A' \subseteq A$, it must be that $V'_x = V_x$.

All of this allows us to conclude that $\texttt{^x} = A(\mathbf{H}(a), T_x) = V'_x(\mathbf{H}(a), T_x) = V_x(\mathbf{H}(a), T_x) = \texttt{* x}$, so K reduces to $K' = \langle \ell' \ \vec{a} \mid E' \mid K \rangle$, preserving the simulation.

□

J Operational Semantics of MiniML

MiniML extends μML with the following continuations

$$K ::= assume \mid ...$$

and the following reductions.

$$\frac{}{\langle\ \texttt{assume \{ e \}} \mid E \mid K\ \rangle \longrightarrow \langle\ e \mid E \mid assume \cdot K\ \rangle}$$

$$\frac{}{\langle\ \texttt{true} \mid E \mid assume \cdot K\ \rangle \longrightarrow \langle\ \texttt{()} \mid E \mid K\ \rangle}$$

$$\frac{}{\langle\ \texttt{false} \mid E \mid assume \cdot K\ \rangle \longrightarrow \langle\ \texttt{false} \mid E \mid assume \cdot K\ \rangle}$$

$$\frac{}{\langle\ \texttt{any} \mid E \mid K\ \rangle \longrightarrow \langle\ \texttt{v} \mid E \mid K\ \rangle}$$

K Example programs run on Proof-of-Concept tool

```
pub struct List {
    val: u32,
    next: Option<Box<List>>,
}

pub fn index_mut(mut l: &mut List, mut ix: usize) -> &mut u32 {
    while ix > 0 {
        match l.next {
            Some(ref mut n) => {
                l = n;
            }
            None => std::process::abort(),
        }
        ix -= 1;
    }

    &mut l.val
}

pub fn write(l: &mut List, ix: usize, val: u32) {
    *index_mut(l, ix) = val;
}
```

```
fn main() {
    let mut l = List {
        val: 1,
        next: Some(Box::new(List {
            val: 10,
            next: None,
        })),
    };
    write(&mut l, 0, 2);
    let l2 = List {
        val: 2,
        next: Some(Box::new(List {
            val: 10,
            next: None,
        })),
    };

    assert(l, l2);
}

#[derive(PartialEq, Eq, Debug)]
pub struct List {
    head: Option<Box<Node>>,
}

#[derive(PartialEq, Eq, Debug)]
pub struct Node {
    val: u32,
    next: Option<Box<Node>>,
}

pub fn rev(l: &mut List) {
    let mut prev = None;
    let mut head = l.head.take();

    while let Some(mut curr) = head {
        let next = curr.next;
        curr.next = prev;
        prev = Some(curr);
        head = next;
    }

    l.head = prev;
}

fn main() {
    let mut l1 = List {
        head: Some(Box::new(Node {
            val: 1,
            next: Some(Box::new(Node {
                val: 10,
                next: None,
            })),
```

```
        })),
    };
    rev(&mut l1);
    let l2 = List {
        head: Some(Box::new(Node {
            val: 10,
            next: Some(Box::new(Node { val: 1, next: None })),
        })),
    };

    assert_eq!(l1, l2);
}
```