TWAM: A Certifying Abstract Machine for Logic Programs

ROSE BOHRER and KARL CRARY, Carnegie Mellon University, USA

Type-preserving (or typed) compilation uses typing derivations to certify correctness properties of compilation. We have designed and implemented a type-preserving compiler for a simply-typed dialect of Prolog we call T-Prolog. The crux of our approach is a new *certifying abstract machine* which we call the Typed Warren Abstract Machine (TWAM). The TWAM has a dependent type system strong enough to specify the semantics of a logic program in the logical framework LF. We present a soundness metatheorem which constitutes a partial correctness guarantee: well-typed programs implement the logic program specified by their type. This metatheorem justifies our design and implementation of a certifying compiler from T-Prolog to TWAM.

CCS Concepts: • **Theory of computation → Logic and verification**; **Constraint and logic programming**; Type theory; • **Software and its engineering → Compilers**;

Additional Key Words and Phrases: Certifying Compilation, Proof-Producing Compilation, Prolog, Warren Abstract Machine

ACM Reference Format:
Rose Bohrer and Karl Crary. 2017. TWAM: A Certifying Abstract Machine for Logic Programs. *ACM Trans. Comput. Logic* 1, 1, Article 1 (April 2017), 41 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Compiler verification is important because of the central role that compilers play in computing infrastructure, and because compiler bugs are easy to make but often difficult to catch. Most work on compiler verification has been done in the setting of imperative or functional programming; very little has been done for logic programming languages like Prolog.

Compiler verification is an equally interesting problem in the case of logic programming. Logic programs are often easier to write correctly than programs in other paradigms, because a logic program is very close to being its own specification. However, the correctness advantages of logic programming cannot be fully realized without compiler verification. Compiler correctness is a concern for logic programming given the scale of realistic language implementations; for example, SWI-Prolog is estimated at over 500,000 lines of code [26].

Certifying compilation [18] is an approach to verification wherein a compiler outputs a formal proof that the compiled program satisfies some desired properties. Certifying compilation, unlike conventional verification, has the advantage that the certificates can be distributed with the compiled code and checked independently by third parties, but the flip side is that compiler bugs are not found until the compiler sees a program that elicits the bug. In the worst case, bugs might be found by the compiler's users, rather than its developers.

Authors' address: Rose Bohrer; Karl Crary, Carnegie Mellon University, Computer Science Department, Pittsburgh, PA, 15213, USA.

1529-3785/2017/4-ART1 $15.00
https://doi.org/0000001.0000001

In most work on certifying compilation [18], an additional disadvantage is that while type and memory safety are certified, dynamic correctness is not. In contrast, we certify the dynamic behavior of logic programs, whose semantics we give as a signature in the logical framework LF [10]. This semantics abstracts away the low-level operational details of Prolog semantics such as order of execution. This brings the type system into close harmony with our source programs, allowing type correctness to naturally encompass dynamic correctness.

In this work, we develop the Typed Warren Abstract Machine (TWAM), a dependently-typed *certifying abstract machine* suitable as a compilation target for certifying compilers. Section 6 formalizes and proves the following claim that TWAM certifies partial correctness:

Theorem 1(*Soundness*): If a query `?-`G. succeeds, there exists a proof of G in LF. That is, well-typed TWAM programs satisfy partial correctness with respect to their LF semantics. Because this theorem says that well-typed TWAM programs implement sound proof search procedures for the LF specification, we also call this theorem *soundness*. We show that TWAM is a suitable compilation target by implementing a compiler from a simply-typed dialect of Prolog called T-Prolog to the TWAM. The result is a certifying compiler with a small, domain-specific proof checker as its trusted core: the TWAM typechecker.

We ease the presentation of TWAM by first presenting its simply-typed variant (SWAM) in Section 4 along with standard progress and preservation theorems, which show type and memory-safety. We then develop a dependently-typed variant in Section 6 whose type system expresses the behavior of a TWAM program as a logic program in the logical framework LF [10], using an encoding demonstrated in Section 5.

2 SOURCE LANGUAGE: T-PROLOG

Our compiler accepts programs for a simply-typed dialect of Prolog which we named T-Prolog. It is worth noting that the language need not be typed for our approach to work: if we wished to work in an untyped dialect of Prolog, we could simply add a compiler pass to collect a list of all the constructors used in a particular Prolog program and construct a single type called `term` containing every constructor we need. We choose a simply-typed language over an untyped one because our use of LF in the TWAM makes this feature easy to implement and because the correspondence with LF is easier to present for a simply-typed language.

T-Prolog programs obey the following grammar:

programs	P	$::= D^* \; Q$
query	Q	$::= ?\!- t.$
declaration	D	$::= D_\tau \mid D_c \mid D_p$
type declaration	D_τ	$::=$ `ident : type`
constructor declaration	D_c	$::=$ `ident :` τ_c
predicate declaration	D_p	$::=$ `ident :` $\tau_p \; C^*$
constructor types	τ_c	$::=$ `type` $\mid$ `ident ->` τ_c
predicate types	τ_p	$::=$ `prop` $\mid$ `ident ->` τ_p
clause	C	$::= t. \mid t$ `:-` G^*
goals	G	$::= t. \mid t, G^*$
terms	t	$::=$ `Ident` $\mid$ `ident`$(t, \ldots, t)$

As our running example throughout the paper, we consider a series of arithmetic operations on the Peano representation of the natural numbers *zero* and *succ*(n). To start, here is the `plus` function written in T-Prolog, with the query `2 + 2 = X`. As in standard Prolog, we will often annotate constructors and predicates with their arities as in `plus/3`. However, each identifier in T-Prolog

has a unique type and thus a unique arity, so annotating identifiers with their arity is not strictly necessary.

Example 2.1.
```
nat : type.
zero/0 : nat.
succ/1 : nat -> nat.

plus/3 : nat -> nat -> nat -> prop.
plus(zero,X,X).
plus(succ(X),Y,succ(Z)) :-
  plus(X,Y,Z).

?- plus(succ(succ(zero)), succ(succ(zero)), X).
```

There is no fundamental difference between `type` and `prop` (and in the theory they are identical): we differentiate them in the T-Prolog syntax because we find this notation intuitive and because it makes the language easier to parse.

2.1 Semantics of T-Prolog

In order to certify that a compiler preserves the dynamic semantics of T-Prolog programs, we must first ascertain those semantics. As in typical Prolog, a T-Prolog program is defined as a signature of logical inference rules, and execution proceeds via depth-first proof-search under that signature, trying rules in the typical Prolog order. Seeing as Prolog evaluation is proof search, the semantics of Prolog are often given operationally in terms of proof-search trees. This operational treatment has the advantage that it can naturally express non-termination and the order in which rules are tried. The disadvantage is that, in increasing operational detail, we diverge further from the world of pure logic, increasing the difficulty of verification.

For this reason, while the T-Prolog implementation does evaluate in the same order as Prolog, we do not take the operational search-based semantics of T-Prolog as canonical. Rather, we take as the meaning of a T-Prolog program the set of formulas provable from its inference rules (Section 5), without regard to the order in which the proof steps are performed. The abstractions made in this semantics are not significantly greater than those already made by a proof-search semantics. The common insight is that a formal semantics for logic programs should exploit the close relationship to logic, ignoring implementation details that have no logical equivalent. In both semantics, for example, it is typical to ignore Prolog's cut operator `!`, which has the side effect of skipping any remaining backtracking opportunities for the current predicate, typically used as an optimization. The cut operation is inherently about search rather than truth, informing the Prolog implementation to ignore some of the available proof rules.

Both the search semantics and provability semantics implicitly assume that all Prolog terms are finite. The backbone of Prolog proof search is unification, and as usual for unification, finiteness cannot be taken for granted. Allowing instances such as $X = f(X)$ to unify would result in infinite terms that do not have a logical interpretation. In typical Prolog implementations, such terms are accepted out of the interest of performance. In T-Prolog, we apply the standard solution of using an occurs check in unification, causing unification to fail on instances such as $X = f(X)$ whose only solutions are infinite. This restores the close correspondence with logic, at the cost of decreased performance.

3 THE TWAM INSTRUCTION SET

The TWAM borrows heavily from the Warren Abstract Machine, the abstract machine targeted by most Prolog implementations [25]. For a thorough, readable description of the WAM, see Aït-Kaci [1]. Readers familiar with the WAM may wish to skim this section and observe the differences from the standard WAM, while readers unfamiliar with the WAM will wish to use this section as a primer or even consult Aït-Kaci's book. In this section we present our simplified instruction set for the WAM using examples. Notable simplifications include the usage of continuation-passing style and omission of many optimizations (with the exception of tail-call optimization in Section 6.5) in order to simplify the formalism. The description here is informal; the formal semantics are given in Section 4.5.

Prolog and WAM Terminology. The following terminology will be used extensively in this paper to describe Prolog and the WAM: a *Prolog term* is an arbitrary combination of *unification variables* X combined with *constructors* such as *succ* and *zero*. What we call *constructors* are generally called *functors* in Prolog terminology. We use the phrase unification variable when discussing Prolog source text and instead use *free variable* to discuss WAM state at runtime. The distinction becomes significant, e.g. because the Prolog source may specify that a parameter to some predicate is a unification variable, but at runtime the argument is a ground term. We use the word *constructor* only when discussing data and use the word *predicate* to refer both to predicates for which a WAM program implements proof search and to the implementation itself. We also say that certain WAM instructions are constructors because they construct some Prolog term, or destructors if they perform pattern matching on some Prolog term. A *structure* is the WAM representation of a constructor applied to its arguments. A predicate consists of one or more *clauses*, each of which specifies one inference rule and each of which consists of a *head term* along with zero or more *subgoals*. A user interacts with the Prolog program by making a *query*, which is compiled in the same way as a predicate with one clause with one subgoal. In our discussion of TWAM programs, we consider programs with arbitrarily many predicates, one of which is designated as the query.

Term Destructors. The instructions `get_var`, `get_val`, and `get_str` are used the implementation of predicates to destruct the predicate arguments:

- `get_var` r_d, r_s reads (gets) r_s into r_d. This is an unconditional register-to-register move and thus its use can be minimized by good register allocators. This is used to implement clauses where a unification variable is an argument.
- `get_val` r_1, r_2 reads (gets) r_1 and r_2 and unifies their values against each other. This is used to implement clauses where multiple arguments are the same unification variable.
- `get_str` r_s, c reads (gets) r_s and unifies it against the constructor c. For our initial examples, we will consider only the case where c has no arguments. `get_str` is effectively an optimized special-case of `get_val` where we know the second unificand must be c. This is used to implement clauses where a constructor appears as a predicate argument.

For example, the Prolog predicate `both_zero(zero, zero)`, which holds exactly when both arguments are zero, could be compiled in all of the following ways, with the naming convention that the register for argument i is named A_i and the i'th temporary is named X_i:

Example 3.1 (Implementing a Predicate)**.**

```
# Implementation 1
get_str A_1, zero/0;
get_str A_2, zero/0;
```

```
# Implementation 2
get_str A_1, zero/0;
 get_val A_1, A_2;
```

```
# Implementation 3
get_var X_1, A_1;
get_var X_2, A_2;
get_str X_1, zero/0;
 get_val X_1, X_2;
```

Generally speaking, Implementation 1 is most efficient, then Implementation 2, then Implementation 3. Note that even though the Prolog predicate `both_zero(zero, zero)` contains no unification variables, we can still use `get_val` in the implementation, because the unification problems $A_1 = zero, A_2 = zero$ and $A_1 = zero, A_1 = A_2$ are equivalent. Observe that any instruction that uses unification, such as `get_val` and `get_str`, will fail if unification fails. Should this occur, the runtime automatically backtracks if possible; backtracking is *never* executed explicitly in the text of a TWAM program.

Term Constructors and Jumps. To implement a query or subgoal that uses the predicate `both_zero`, we must first construct its arguments, then jump to the implementation:

- `put_var` r_d writes (puts) a *new* free variable into r_d. This is used to implement passing a unification variable as an argument.
- `put_val` r_d, r_s writes (puts) the value of an *existing* unification variable into r_d, assuming it is already in r_s. This is an unconditional register move. Thus it is entirely identical to `get_var`. For this reason, in our theory we will condense these into one instruction `mov` r_d, r_s and only use the names `get_var` and `put_val` for consistency with traditional terminology in our examples.
- `put_str` r_d, c writes a structure into r_d using constructor c. `get_str` is effectively an optimized special-case of `put_val` where we are storing not an arbitrary unification variable, but specifically a constant c. This is used to implement passing a constructor as an argument to a predicate.
- `jmp` ℓ^C passes control to the code location (address literal) ℓ^C. Arguments are passed through registers. All code is in continuation passing style, and thus a continuation can be passed in through a register, which is named `ret` by convention. The queries in Example 3.2 do not require returning from predicate calls, thus continuations are discussed separately.

Example 3.2 (Making a Query)**.**

```
# both_zero(X, X).
put_var A_1;
put_val A_2, A_1;
jmp both_zero/2;
```

```
# both_zero(X, zero).
put_var A_1;
put_str A_2, zero/0;
jmp both_zero/2;
```

Constructors with Arguments. We continue to use the `put_str` and `get_str` instructions to construct and destruct structures that contain arguments. The difference is that when calling `put_str` or `get_str` with a constructor of arity $n > 0$, we now initiate a *spine* (terminology ours) consisting of n additional instructions using only the following:

- When `unify_var` r is the i'th instruction of a spine, it unifies the i'th argument of the constructor with a *new* unification variable, at register r.
- When `unify_val` r is the i'th instruction of a spine, it unifies the i'th argument of the constructor with an *existing* unification variable, at register r.

The same instructions are used with both `put_str` and `get_str` spines. However, at runtime a spine will execute in one of two modes, *read mode* or *write mode*. *Read mode* is used to destruct an existing value, meaning we are in the `get_str` r_s, c and r_s contains a structure whose constructor is c. *Write mode* is used to construct a new value, meaning we are either in `put_str` r_d, c or we are in `get_str` r_s, c but the content of r_s is a free variable. In both modes, each unification instruction processes one constructor argument:

- Read-mode `unify_var` stores the next constructor argument in a register.
- Read-mode `unify_val` unifies the next constructor argument with the content of a register.

- Write-mode `unify_var` allocates a free variable as constructor argument, storing it also in a register.
- Write-mode `unify_val` uses the content of a register as constructor argument.

For example, the Prolog predicate `same_pos(succ(X), succ(X))` which holds when the arguments are the same positive number, can be implemented and used as follows:

Example 3.3 (Predicates with Prolog Spines)**.**

```
# Implementation
get_str A_1, succ/1;
  unify_var X_1;
get_str A_2, succ/1;
  unify_val X_1;
```

```
# Query same_pos(succ(X),succ(X))
put_str A_1, succ/1;
  unify_var X_1;
put_str A_2, succ/1;
  unify_val X_1; # Reads X_1
jmp same_pos/2;
```

```
# Query same_pos(succ(X),succ(Y))
put_str A_1, succ/1;
  unify_var X_1;
put_str A_2, succ/1;
  unify_var X_1; # Overwrites X_1
jmp same_pos/2;
```

```
# Query same_pos(zero,succ(succ(zero)))
put_str A_1, zero/0;
put_str X_1, zero/0; # Z = 0
put_str X_2, succ/1; # Y = 1
  unify_val X_1;
put_str A_2, succ/1; # X = 2
  unify_val X_2;
jmp same_pos/2;
```

The last example demonstrates a compilation technique known as *flattening*: The unification problem $X = succ(succ(zero))$ is equivalent to the problem $X = succ(Y), Y = succ(Z), Z = zero$. This allows us to implement nested structures such as $succ(zero)$ or $succ(succ(succ(zero)))$ by introducing intermediate variables. Thus each spine need only introduce one structure, and nested structures are reduced to the one-structure case by flattening.

Continuations, Closures, and Halting. Prolog proof search can be structured using success and failure continuations. [9] When a predicate has multiple clauses, failure continuations are used to remember alternate clauses and implement backtracking. When a clause has multiple subgoals, success continuations are used to remember the remaining subgoals. In our system, success continuations can be stored in registers and passed to predicates, typically in a register named `ret`, whereas failure continuations are stored in the trail. Both success and failure continuations can access an environment value (generally a tuple) through the register `env`. Tuples are like structures, but can contain closures and cannot be unified. The entry-point of a TWAM program is a top-level query, which specifies an initial continuation that terminates the program in success. If all clauses fail, then the runtime will automatically report that the program failed.

- `close` r_d, r_e, ℓ^C places a new closure in r_d containing an environment read from r_e. When that closure is invoked, control will pass to ℓ^C and the environment will be placed in a special-purpose register named `env`. This is used to construct success continuations.
- `push_bt` r_e, ℓ^C (push backtracking point) creates a new failure continuation. When that continuation is invoked, control will pass to ℓ^C and the environment will be placed in `env`. Note that `push_bt` does not take a destination register: a stack of failure continuations is stored implicitly in the machine state, and they are only ever invoked implicitly, when unification instructions like `get_val` fail.
- `put_tuple` r_d, n begins a *tuple spine* of length n which will put a new tuple in r_d. All following instructions of the tuple spine are `set_val`.
- `set_val` r_s copies r_s in as the next tuple element.

- `proj` r_d, r_s, i copies the i'th element of the tuple at r_s into r_d.
- `succeed` immediately terminates the program and indicates that the initial query has succeeded. (At this point, the runtime system will print out the solution to the query.)

As an example, consider implementing and calling the predicate $X + Y = Z$ with two clauses: `plus(zero,X,X)` and `plus(succ(X),Y,succ(Z)) :- plus(X,Y,Z)`:

Example 3.4 (Implementing `plus`)**.**

```
# Entry point to plus, implements the
# case plus(zero,X, X) and tries
# plus-succ on failure
plus-zero/3:
  put_tuple X_1, 3;
    set_val A_1;
    set_val A_2;
    set_val A_3;
  push_bt X_1, plus-succ/3;
  get_str A_1, zero/0;
  get_val A_2, A_3;
  jmp ret;
```

```
# plus(succ(X), Y, succ(Z)) :- plus(X,Y,Z).
plus-succ/3:
  proj A_1, env, 1;
  proj A_2, env, 2;
  proj A_3, env, 3;
  get_str A_1, succ/1;
    unify_var A_1;
  get_str A_3, succ/1;
    unify_var A_3;
  jmp plus-zero/3;
```

Example 3.5 (Calling `plus`)**.**

```
init-cont/0:
  succeed;
```

```
# plus(succ(zero), succ(zero), X)
query/0:
  put_tuple X_1, 0;
  close ret, X_1, init-cont/0;
  put_str X_2, zero/0;
  put_str A_1, succ/1;
    unify_val X_2;
  put_str A_2, succ/1;
    unify_val X_2;
  put_var A_3;
  jmp plus-zero/3;
```

In this example, `plus-zero/3` is the entry point for addition, and implements the base case. Because `plus-zero/3` is not the last case, it constructs a failure continuation which tries the `plus-succ/3` case if an instruction fails. This requires remembering the environment, implemented by creating a tuple. In the example query, the first invocation of `plus-zero/3` will fail on the `get_str` instruction because A_1 contains $succ(zero)$, not $zero$, causing `plus-succ/3` to run (which will succeed after another call to `plus-zero`). `plus-succ` contains several optimizations. The final subgoal of a clause can always apply tail-call optimization, so no success continuation is necessary. Furthermore, it carefully avoids the use of intermediate registers. For example, when reading the argument $succ(X)$, the variable X is written directly into A_1 to prepare for the recursive call. The query $plus(succ(zero), succ(zero), X)$ must specify an initial continuation, which simply reports success. Because the success continuation is trivial, the empty tuple suffices as its environment.

Runtime State. The runtime representation of a TWAM program differs from that of a WAM program, following the differences in their instruction sets. Both languages have a fixed *code section* containing the TWAM program text and a variable-sized *heap*, which maintains all Prolog terms in a union-find data structure to enable fast unification. The most significant difference is that

the TWAM machine state does not have a stack, but instead allocates success continuations on the heap and allows them to be garbage collected. Failure continuations, however, are stored in a separate area called the *trail* as in standard WAM. In addition to storing a closure created with `push_bt`, the trail automatically keeps track of all state changes which might have to be reverted during backtracking. Traditional descriptions of the WAM contain a *push-down list* or PDL area, which is used in unification to store a temporary list of unification subproblems. Because this data structure is used only during unification, we found it easier to express the PDL merely as a part of unification and not as a permanent part of the state.

Differences Between WAM and TWAM Instruction Sets. The key difference between WAM and TWAM is that the TWAM implements predicate calls and backtracking with success and failure continuations, while WAM implements both by maintaining a custom stack discipline, whose invariants are non-trivial. The use of CPS significantly simplifies the formalism and unifies several instructions that are distinct in traditional WAM:

- Environments in TWAM are expressed as tuples with the instructions `put_tuple`, `set_val`, and `proj`, which replace `allocate` and `deallocate`.
- The `jmp` instruction of TWAM unifies `call`, `execute`, and `proceed` from WAM.
- The `push_bt` instruction of TWAM replaces `try_me_else`, `retry_me_else`, and `trust_me` from WAM.
- The `succeed` instruction is added in TWAM for stylistic purposes; WAM reuses `proceed` for this purpose.
- The unification and spinal instructionns of TWAM correspond directly to WAM.
- TWAM omits several optimizations such as cut and case analysis.

4 THE SIMPLY-TYPED WAM (SWAM)

The core contributions of this work are the design, metatheory, and implementation of a type system for the TWAM strong enough to certify compilation. The certification guarantees provided by the dependently-typed TWAM in Section 6 require significant complexity in the type-system. In this section, we ease the presentation of that system by first presenting its simply-typed variant, the SWAM. We prove progress and preservation for SWAM, which constitute a safety property analogous to those of other strongly-typed abstract machines such as the typed assembly language TAL [16]. In Section 6, this is subsumed by progress and preservation for TWAM, which is strong enough to certify partial dynamic correctness.

4.1 Typechecking SWAM

The text of a SWAM program is structured as a code area C mapping code locations ℓ^C to code values $\mathsf{code}[\Gamma](I)$. A code value is a single basic block I annotated with a *register file type* (rftype) Γ which indicates, for each register r_i, the type expected by I. One of those code values is designated as the query (a predicate with one clause and one subgoal), which is the entry point of the program. The type τ assigned to each register is either an *atomic type* a representing a Prolog term, a *continuation type* $\neg\Gamma$ representing a closure that expects the registers to obey the types in Γ, or a *tuple type* $\mathrm{x}[\vec{\tau}]$, where the elements $\vec{\tau}$ can freely mix atomics and continuations. Here $\vec{\tau}$ is an abbreviation for the sequence $\tau_1, \ldots, \tau_n$; similar abbreviations will be used extensively throughout the paper.

The main typing judgement in SWAM is $\Gamma \vdash_{\Sigma;\Xi} I\ \mathsf{ok}$, which says the basic block I is well-typed assuming the registers obey Γ initially, and given signatures Σ, Ξ which assign types to every constructor c and code location ℓ^C, respectively. We omit the subscripts $\Sigma; \Xi$ on rules where they are not relevant. Throughout the paper, the notation $\Gamma\{r\colon \tau\}$ refers to updating the type of r in

Γ to be τ. Analogous notation will be used for register values, etc. Throughout this section, we alternate between inference rules for typechecking instructions and their descriptions.

$$\frac{}{\Gamma \vdash \texttt{succeed}; I\ \texttt{ok}}\ \textsc{Succeed} \qquad \frac{\Gamma\{r : a\} \vdash I\ \texttt{ok}}{\Gamma \vdash \texttt{put\_var}\ \ [a]r; I\ \texttt{ok}}\ \textsc{PutVar} \qquad \frac{\Gamma(r_1) = a \quad \Gamma(r_2) = a \quad \Gamma \vdash I\ \texttt{ok}}{\Gamma \vdash \texttt{get\_val}\ r_1, r_2; I\ \texttt{ok}}\ \textsc{GetVal}$$

$$\frac{\Gamma \vdash_{\Sigma;\Xi} op : \neg\Gamma' \quad \cdot \vdash \Gamma' \leq \Gamma}{\Gamma \vdash_{\Sigma;\Xi} \texttt{jmp}\ op, I\ \texttt{ok}}\ \textsc{Jmp} \qquad \frac{\Gamma(r_s) = \tau \quad \Gamma\{r_d : \tau\} \vdash I\ \texttt{ok}}{\Gamma \vdash \texttt{mov}\ r_d, r_s; I\ \texttt{ok}}\ \textsc{Mov}$$

$$\frac{\Gamma \vdash I\ \texttt{ok} \quad \Gamma(r_e) = \tau \quad \Xi(\ell^C) = \neg\{\texttt{env}: \tau\}}{\Gamma \vdash_{\Sigma;\Xi} \texttt{push\_bt}\ r_e, \ell^C; I\ \texttt{ok}}\ \textsc{PushBT} \qquad \frac{\Gamma(r_e) = \tau \quad \Gamma\{r_d : \neg\Gamma'\} \vdash I\ \texttt{ok} \quad \Gamma \vdash \ell^C : \neg\Gamma'\{\texttt{env}: \tau\}}{\Gamma \vdash \texttt{close}\ r_d, r_e, \ell^C; I\ \texttt{ok}}\ \textsc{Close}$$

- `succeed` always typechecks, and is typically the last instruction of its block.
- `put_var` $[a]r$ allocates a free variable of type a in r, thus updating r to type a. We write the annotation $[a]$ in brackets to emphasize that it is used only for typechecking.
- `get_val` r_1, r_2 unifies r_1 and r_2, so they must have the same (atomic) type.
- `jmp` op transfers control to op, which in the general case is either a location ℓ^C (used in predicate calls) or register r (used in returns, by convention generally named `ret`). The judgement $\cdot \vdash \Gamma' \leq \Gamma$ means $\forall r \in Dom(\Gamma').\Gamma'(r) = \Gamma(r)$ (Γ' may omit some registers of Γ). The judgement $\Gamma \vdash_{\Sigma;\Xi} op : \tau$ has the rules:

$$\frac{\Xi(\ell^C) = \tau}{\Gamma \vdash_{\Sigma;\Xi} \ell^C : \tau}\ \textsc{Op-}\ell^C \qquad \frac{\Gamma(r) = \tau}{\Gamma \vdash_{\Sigma;\Xi} r : \tau}\ \textsc{Op-}r$$

- `mov` r_d, r_s copies r_s into r_d.
- `push_bt` r_e, ℓ^C installs the failure continuation ℓ^C in the trail along with the environment from r_e, which will be in `env` upon invocation of ℓ^C.
- `close` r_d, r_s, ℓ^C is analogous, but stores the resulting success continuation in r_d before proceeding.

$$\frac{\Gamma(r_s) = \mathrm{x}[\vec{\tau}] \quad \Gamma\{r_d : \tau_i\} \vdash I\ \texttt{ok} \quad (\text{where } i \leq |\vec{\tau}|)}{\Gamma \vdash \texttt{proj}\ r_d, r_s, i; I\ \texttt{ok}}\ \textsc{Proj} \qquad \frac{\Gamma \vdash I :_t (\vec{\tau} \to \{r_d : \mathrm{x}[\vec{\tau}]\}) \quad (\text{where } n = |\vec{\tau}|)}{\Gamma \vdash \texttt{put\_tuple}\ r_d, n; I\ \texttt{ok}}\ \textsc{PutTuple}$$

- `proj` r_d, r_s, i puts the i'th element the tuple r_s into r_d, typechecking only if r_d has length at least i. Here $\vec{\tau}$ is a sequence of types, one for each element.
- `put_tuple` r_d, n initiates a *tuple spine* of length n with destination r_d. The remainder of the tuple spine is checked using the auxiliary *tuple spine typing* judgement $\Gamma \vdash I :_t (\tau_1 \to \cdots \to \tau_n \to \texttt{Post})$, where `Post` is a singleton rftype $\{r_d : \mathrm{x}[\vec{\tau}]\}$. The auxiliary judgement $\Gamma \vdash I :_t (\tau_1 \to \cdots \to \tau_n \to \texttt{Post})$ should be read as "the next n instructions construct tuple elements of type τ_i, with postcondition $\texttt{Post} \leq \Gamma$, and all remaining instructions typecheck". The typing rules for the spine typing judgement are given in Section 4.2.

$$\frac{\Sigma(c) = \vec{a} \to a \quad \Gamma(r) = a \quad \Gamma \vdash I :_s (\vec{a} \to \{\})}{\Gamma \vdash \texttt{get\_str}\ c, r; I\ \texttt{ok}}\ \textsc{GetStr} \qquad \frac{\Sigma(c) = \vec{a} \to a \quad \Gamma\{r : a\} \vdash I :_s (\vec{a} \to \{\})}{\Gamma \vdash \texttt{put\_str}\ c, r; I\ \texttt{ok}}\ \textsc{PutStr}$$

- `get_str` c, r and `put_str` c, r both initiate *Prolog spines* which are checked with *Prolog spine typing* judgement $\Gamma \vdash I :_s (a_1 \to \cdots \to a_n \to \texttt{Post})$. Unlike tuple spines, Prolog spines contain only atomic types, and in SWAM always have an empty postcondition $\texttt{Post} = \{\}$. Intuitively one might expect $\texttt{Post} = \{r : a\}$, for `put_str`, but we choose to update the type of r at the *beginning* of the spine instead of the end, because this leaves `put_str` symmetric more symmetric with `get_str` (a free variable is stored at r_d until the spine completes to ensure type safety).

4.2 Spine Typing

When constructing compound data structures (either tuples or structures), we wish to know that the data structure has the intended number of arguments, each with the intended type. For this reason, we apply the auxilliary typing judgements for tuple and Prolog spines. Each spinal instruction produces one element, and so each rule application checks the type of one element. Consider the rules for the tuple spine judgement $\Gamma \vdash I :_t (\tau_1 \to \cdots \to \tau_n \to \texttt{Post})$:

$$\frac{\Gamma(r) = \tau \quad \Gamma \vdash I :_t J}{\Gamma \vdash \texttt{set\_val}\ r; I :_t (\tau \to J)}\ \textsc{TSpine-SetVal} \qquad \frac{(\Gamma{+}\texttt{Post}) \vdash \texttt{I ok}}{\Gamma \vdash I :_t \texttt{Post}}\ \textsc{TSpine-End}$$

The rule for (TSpine-SetVal) says that each `set_val` contributes one element. The rule (TSpine-End) resumes the main typing judgement $\Gamma \vdash I$ ok when says that when a tuple is complete, we store the tuple according to `Post` and resume normal typechecking. Specifically, $(\Gamma{+}\Gamma')$ is the rftype such that $(\Gamma{+}\Gamma')(r) = \Gamma'(r)$ for $Dom(\Gamma')$ and $(\Gamma{+}\Gamma')(r) = \Gamma(r)$ otherwise.

Prolog spines have their own auxilliary judgment, $\Gamma \vdash I :_s (a_1 \to \cdots \to a_n \to \texttt{Post})$. The rule for ending a Prolog spine is analogous to (TSpine-End). The elements of a Prolog spine can be produced either by `unify_val` r or `unify_var` r. The `unify_val` r instruction which requires the argument register type to match the constructor argument, whereas `unify_var` r creates a new unification variable of the correct type, which appears both in r_d and as a constructor argument.

$$\frac{\Gamma(r) = a \quad \Gamma \vdash I :_s J}{\Gamma \vdash \texttt{unify\_val}\ r; I :_s (a \to J)}\ \textsc{UnifyVal} \qquad \frac{\Gamma\{r\colon a\} \vdash I :_s J}{\Gamma \vdash \texttt{unify\_var}\ r; I :_s (a \to J)}\ \textsc{UnifyVar}$$

4.3 State Representation and Invariants

Following the traditional description of the WAM, the essential parts of the SWAM *machine state* include the *code section* C, *heap* H, and *trail* T (backtracking structure). H and C are often considered together as the *store* $S = (C, H)$. Locations in the code section are written ℓ^C and locations in the heap are written ℓ^H. Where both are acceptable we write ℓ. The notation $S(\ell)$ denotes either $H(\ell^H)$ or $C(\ell^C)$ as appropriate. We additionally have an explicit representation of the *register file* R and we represent the instruction pointer as the sequence I of remaining instructions in the current basic block. Machines also support three spinal execution modes: read spines, write spines, and tuple (write) spines. In short, machine states are described by the syntax:

$$m ::= (T, S, R, I) \mid \texttt{read}(T, S, R, I, \vec{\ell}^H) \mid \texttt{write}(T, S, R, I, c, \ell^H, \vec{\ell}^H) \mid \texttt{twrite}(T, S, R, I, r, n, \vec{\ell}^H)$$

We first consider the following typing invariant for normal states (T, S, R, I) in depth and then revisit the additional invariants for spinal states:

$$\frac{\begin{array}{c} S \vdash T\ \texttt{ok} \\ \cdot \vdash S : (\Xi; \Psi) \quad \Psi \vdash R\colon \Gamma \quad \Gamma \vdash I\ \texttt{ok} \end{array}}{\cdot \vdash_{\Sigma;\Xi} (T, S, R, I)\ \texttt{ok}}\ \textsc{Mach}$$

As in Section 4.1, all judgments are parameterized by signatures Σ and code section types Ξ, which are elided when irrelevant. The code section is well-typed when each basic block is well-typed according to the rules of Section 4.1. The code section is allowed to be mutually recursive:

$$\frac{\cdot \vdash_{\Sigma;\Xi} v_1^C\colon \tau_1 \ \cdots\ \cdot \vdash_{\Sigma;\Xi} v_n^C\colon \tau_n \quad (\text{where } \Xi = \{v_1^C\colon \tau_1, \ldots, v_n^C\colon \tau_n\})}{\cdot \vdash_{\Sigma;\Xi} \{v_1^C, \ldots v_n^C\}\colon \Xi}\ \textsc{Code-Sec}$$

Heap types are written Ψ and are analogous to rftypes. As in rftypes we write $\Psi\{\ell^H\colon \tau\}$ when updating the type of ℓ^H. We also write $\Psi\{\{\ell^H\colon \tau\}\}$ when adding a *fresh* location ℓ^H with type τ, or $\{\}$ for an empty heap or empty heap type. We prohibit cycles in the heap because it simplifies implementing SWAM and simplifies the dependent type system of Section 6 even further.

Specifically, a typing derivation $\mathcal{D} : (\cdot \vdash H : \Psi)$ serves as a witness that H is acyclic, because $\mathcal{D}$ implicitly specifies a topological sorting on H: the rules below state that each value may only refer to preceding values. However, Ψ need not assign a type to all values in H, so long as those values without types are never accessed. This technicality is useful when reasoning about backtracking as in Lemma 10.

$$\frac{}{\cdot \vdash H : \{\}}\ \textsc{Heap-Nil} \qquad \frac{\cdot \vdash H : \Psi \quad H(\ell^H) = v^H \quad \Psi \vdash v^H : \tau \quad \ell^H \notin Dom(\Psi)}{\cdot \vdash H : \Psi\{\{\ell^H : \tau\}\}}\ \textsc{Heap-Cons}$$

Values are divided into *heap values* v^H which are arbitrarily large and *word values* w which are fixed size. In SWAM, words are always heap locations $w ::= \ell^H$. The heap values v^H follow the syntax:

$$v^H ::= c\langle \ell_1^H, \ldots, \ell_n^H \rangle \mid \textbf{FREE}[a] \mid \textbf{BOUND}\ \ell^H \mid \texttt{close}\,(w_{env}, \ell^C) \mid \langle w_1, \ldots, w_n \rangle$$

The values $c\langle \ell_1^H, \ldots, \ell_n^H \rangle$ and $\textbf{FREE}[a]$ introduce structures and free variables in Prolog terms, respectively. The type annotation a in $\textbf{FREE}[a]$ is merely a convenience for the metatheory and not used at runtime (i.e. SWAM and TWAM support type erasure).

Combined with pointers $\textbf{BOUND}\ \ell^H$, these values provide a union-find data structure within the heap, used by SWAM's unification algorithm. The $\textbf{BOUND}\ \ell^H$ pointers are merely an artifact of that algorithm and are semantically equivalent to ℓ^H. In addition to Prolog terms, the heap contains closures $\texttt{close}\,(w_{env}, \ell^C)$ where the machine word w_{env} is the environment for executing ℓ^C, as well as tuples $\langle w_1, \ldots, w_n \rangle$. The typing invariants for heap values are:

$$\frac{\Psi \vdash w_{env} : \tau \quad \Psi \vdash \ell^C : \neg\Gamma\{\texttt{env}: \tau\}}{\Psi \vdash \texttt{close}\,(w_{env}, \ell^C) : \neg\Gamma}\ \textsc{HV-Close} \qquad \frac{\Psi \vdash w_1 : \tau_1 \quad \cdots \quad \Psi \vdash w_n : \tau_n}{\Psi \vdash \langle w_1, \ldots, w_n \rangle : \texttt{x}[\tau_1, \ldots, \tau_n]}\ \textsc{HV-Tup}$$

$$\frac{\Psi \vdash \ell^H : a}{\Psi \vdash \textbf{BOUND}\ \ell^H : a}\ \textsc{HV-Bound} \qquad \frac{\Sigma(c) = \vec{a} \to a \quad \Psi \vdash \ell_i^H : a_i}{\Psi \vdash_{\Sigma;\Xi} c\langle \ell_1^H, \ldots, \ell_n^H \rangle : a}\ \textsc{HV-Str} \qquad \frac{}{\Psi \vdash \textbf{FREE}[a] : a}\ \textsc{HV-Free}$$

Register Typing. A register file R simply maps registers r_i to word values w_i and is well-typed when all w_i are well-typed. Because the only word values in SWAM are heap locations, it suffices to consult the heap type Ψ:

$$\frac{\Psi \vdash w_1 : \tau_1 \quad \cdots \quad \Psi \vdash w_n : \tau_n}{\Psi \vdash \{r_1 \mapsto w_1, \ldots, r_n \mapsto w_n\} : \{r_1 : \tau_1, \ldots, r_n : \tau_n\}}\ \text{RF} \qquad \frac{\Psi(\ell^H) = \tau}{\Psi \vdash \ell^H : \tau}\ \text{WV-}\ell^H$$

Trail Typing. When Prolog backtracks because a clause failed, it must revert all changes made by the failed clause. The only such change is the binding of free variables during unification, thus it suffices to record bindings when they occur and revert them during backtracking. The *trail* is the data structure that records these variables. In traditional presentations of the WAM, the trail contains variable addresses only and separate *choice point* records in the call stack contain the failure continuation. For our presentation, it simplified the formalism to store the continuation inline. The trail is given as a list of *trail frames* (t, w_{env}, ℓ^C) where t is a list of heap locations (in the theory, annotated with types as in $\ell^H : a$), where we write the list of t_i as $t_1 :: \cdots :: t_n :: \epsilon$. The environment is w_{env} and ℓ^C is the failure continuation. The function $\texttt{unwind}(S, t)$ describes the process of backtracking one trail frame, i.e. $\texttt{unwind}(S, (\ell^H : a) :: t) = \texttt{unwind}(S\{\ell^H \mapsto \textbf{FREE}[a]\}, t)$ and $\texttt{unwind}(S, \epsilon) = S$.

$$\frac{}{S \vdash \epsilon\ \texttt{ok}}\ \textsc{Trail-Nil} \qquad \frac{\begin{array}{c}\texttt{unwind}(S, t) = S' \quad S' \vdash T\ \texttt{ok} \\ \vdash S' : (\Xi, \Psi') \quad \Psi' \vdash w_{env} : \tau \quad \Psi' \vdash \ell^C : \neg\{\texttt{env}: \tau\}\end{array}}{S \vdash_{\Sigma;\Xi} (t, w_{env}, \ell^C) :: T\ \texttt{ok}}\ \textsc{Trail-Cons}$$

Special Mode Invariants. When the machine is in read or write mode, it maintains additional data. Read mode maintains a list of arguments not yet read, while the write modes maintain lists of arguments written so far with destination registers or locations. Prolog write mode also tracks the constructor being applied while tuple write mode tracks the number of elements left to be written in the tuple. In each case additional invariants are required, as given in the judgements $\Psi \vdash \vec{\ell}^H$ `reads` $\vec{a}, \Psi \vdash (\vec{\ell}^H, \ell^H, c)$ `writes` $(\vec{a}_2 \to \{\}) \text{ and } \Psi \vdash (n, r, \vec{\ell}^H)$ `writes` $(\vec{\tau}_2 \to \{r : \mathrm{x}[\vec{\tau}_1 \vec{\tau}_2]\})$. In each case the invariants ensure that the type of the constructor or tuple in question is consistent with both the values computed so far and the remaining spinal instructions.

$$\frac{S \vdash T\ \mathsf{ok} \quad \vdash S : (\Xi; \Psi) \qquad \Psi \vdash R : \Gamma \quad \Gamma \vdash I :_s J \quad \Psi \vdash \vec{\ell}^H \ \texttt{reads}\ J}{\cdot \vdash_{\Sigma;\Xi} \mathsf{read}(T, S, R, I, \vec{\ell}^H)\ \mathsf{ok}}\ \textsc{Mach-Read} \qquad \frac{\Psi \vdash \ell_i^H : a_i}{\Psi \vdash \vec{\ell}^H \ \texttt{reads}\ (\vec{a} \to \{\})}\ \textsc{Reads}$$

$$\frac{S \vdash T\ \mathsf{ok} \quad \cdot \vdash S : (\Xi; \Psi) \qquad \Psi \vdash R : \Gamma \quad \Gamma \vdash I :_s J \quad \Psi \vdash (\vec{\ell}^H, \ell, c)\ \texttt{writes}\ J}{\cdot \vdash_{\Sigma;\Xi} \mathsf{write}(T, S, R, I, c, \ell^H, \vec{\ell}^H)\ \mathsf{ok}}\ \textsc{Mach-Write} \qquad \frac{\Sigma(c) = \vec{a}_1 \to \vec{a}_2 \to a \quad \Psi \vdash \vec{\ell}^H : \vec{a}_1 \quad \Psi(\ell^H) = a}{\Psi \vdash_{\Sigma;\Xi} (\vec{\ell}^H, \ell^H, c)\ \texttt{writes}\ (\vec{a}_2 \to \{\})}\ \textsc{Writes}$$

$$\frac{S \vdash T\ \mathsf{ok} \quad \vdash S : (\Xi; \Psi) \qquad \Psi \vdash R : \Gamma \quad \Gamma \vdash I :_t J \quad \Psi \vdash (\vec{w}, r, n)\ \texttt{writes}\ J}{\cdot \vdash_{\Sigma;\Xi} \mathsf{twrite}(T, S, R, I, \vec{w}, r, n)\ \mathsf{ok}}\ \textsc{Mach-TWrite} \qquad \frac{\Psi \vdash \vec{\ell}^H : \vec{\tau}_1 \quad |\vec{\tau}_2| = n}{\Psi \vdash (n, r, \vec{\ell}^H)\ \texttt{writes}\ (\vec{\tau}_2 \to \{r : \mathrm{x}[\vec{\tau}_1 \vec{\tau}_2]\})}\ \textsc{TWrites}$$

4.4 Operational Semantics

The dynamic semantics of SWAM are given as a small-step operational semantics. We begin with an informal example trace executing the query `?- plus(X,zero,succ(zero))` using the `plus` function of Example 3.4 before developing the semantics formally.

For each line we describe any changes to the machine state, i.e. the heap, trail, register file, and instruction pointer. As with the WAM, the TWAM supports special execution modes for spines: *read mode* and *write mode*. When the program enters read mode, we annotate that line with the list ℓ^Hs of variables being read, and when the program enters write mode we annotate it with the constructor c being applied, the destination location ℓ^H and the argument locations $\vec{\ell}^H$. The final instruction of a write-mode spine is best thought of two evalution steps, one of which constructs the last argument of the constructor and one of which combines the arguments into a term.

Code	Change
`# Query plus(`X`, zero, succ(zero))`	
`query` $\mapsto$ `code[{}](`	
`  put_var` r_1`;`	$H \leftarrow H\{\{\ell_1 \mapsto \mathbf{FREE}[\mathsf{nat}]\}\}, R \leftarrow R\{r_1 \mapsto \ell_1\}$
`  put_str` r_2`, zero/0;`	$H \leftarrow H\{\{\ell_2 \mapsto \mathbf{FREE}[\mathsf{nat}]\}\}, R \leftarrow R\{r_2 \mapsto \ell_2\}, c = \mathtt{zero}$
	$\ell = \ell_2, \vec{\ell} = \langle\rangle,$
	$H \leftarrow H\{\ell \mapsto c\langle\vec{\ell}\rangle\}$
`  put_str` r_3`, succ/1;`	$H \leftarrow H\{\{\ell_3 \mapsto \mathbf{FREE}[\mathsf{nat}]\}\}, R \leftarrow R\{r_3 \mapsto \ell_3\}, c = \mathtt{succ}$
	$\ell = \ell_3, \vec{\ell} = \langle\rangle$
`    unify_val` r_2`;`	$\vec{\ell} \leftarrow \langle\ell_2\rangle, H \leftarrow H\{\ell \mapsto c\langle\vec{\ell}\rangle\}$
`  put_tuple` r_4`, 0;`	$H \leftarrow H\{\{\ell_4 \mapsto \langle\rangle\}\}, R \leftarrow R\{r_4 \mapsto \ell_4\}$
`  close ret,` r_4`, success/0;`	$H \leftarrow H\{\{\ell_5 \mapsto \mathsf{close}\,(\ell_4, \mathsf{success})\}\}, R \leftarrow R\{\mathsf{ret} \mapsto \ell_5\}$
`  jmp plus-zero/3;`	$I \leftarrow C(\mathtt{plus\text{-}zero})$
`)`	
`plus-zero/3` $\mapsto$ `code[`$r_2 : \mathsf{nat}, r_3 : \mathsf{nat}, r_4 : \neg\{\}$`](`	
`  put_tuple` r_4`, 4;`	$\vec{\ell} = \langle\rangle$
`    set_val` r_2`;`	$\vec{\ell} = \langle\ell_1\rangle$
`    set_val` r_2`;`	$\vec{\ell} = \langle\ell_1, \ell_2\rangle$
`    set_val` r_3`;`	$\vec{\ell} = \langle\ell_1, \ell_2, \ell_3\rangle$

```
    set_val ret;              ℓ⃗ = ⟨ℓ1, ℓ2, ℓ3, ℓ5⟩
                              H ← H{{ℓ6 ↦ ⟨ℓ⃗⟩}}, R ← R{r4 ↦ ℓ6}
  push_bt r4, plus-succ/3;    T ← (ℓ6, plus-succ/3, nil) :: nil
  get_str r2, zero/0;         c = zero, l = ℓ1, ℓ⃗ = ⟨⟩
                              H ← H{ℓ1 ↦ zero}, T ← (ℓ6, plus-succ/3, ℓ1) :: ⟨⟩
# This instruction fails, backtrack to plus-succ/3
  get_val r2, r3;             T ← nil, I ← plus-succ/3, H ← H{ℓ1 ↦ FREE[nat]}
  jmp ret;
)

plus-succ/3 ↦ code[{env : x[nat, nat, nat, ¬{}]}] (
  proj r1, env, 1;            R ← R{r1 ↦ ℓ1}
  proj r2, env, 2;            R ← R{r2 ↦ ℓ2}
  proj r3, env, 3;            R ← R{r3 ↦ ℓ3}
  proj ret, env, 4;           R ← R{ret ↦ ℓ5}
# Here we are replacing a free variable with a concrete term
  get_str succ/1, r2;         ℓ = ℓ1, ℓ⃗ = ⟨⟩
    unify_var r2;             H ← H{{ℓ4 ↦ FREE[nat]}}, R ← R{r2 ↦ ℓ4}, ℓ⃗ = ⟨ℓ4⟩
                              H ← H{ℓ1 ↦ succ ℓ⃗},
  get_str succ/1, r3;         ℓ⃗ = ⟨ℓ2⟩
    unify_var r3;             R ← R{r3 ↦ ℓ2},
                              H ← H{ℓ1 ↦ succ ℓ⃗},
  jmp plus-zero/3;            I ← C(plus-zero)
)

plus-zero/3 ↦ code[{r2 : nat, r2 : nat, r3 : nat, r4 : ¬{}}](
  put_tuple r4, 4;            ℓ⃗ = ⟨⟩
    set_val r2;               ℓ⃗ = ⟨ℓ1⟩
    set_val r2;               ℓ⃗ = ⟨ℓ1, ℓ2⟩
    set_val r3;               ℓ⃗ = ⟨ℓ1, ℓ2, ℓ3⟩
    set_val ret;              ℓ⃗ = ⟨ℓ1, ℓ2, ℓ3, ℓ5⟩
                              H ← H{{ℓ7 ↦ ⟨ℓ⃗⟩}}, R ← R{r4 ↦ ℓ6}
  push_bt r4, plus-succ/3;    T ← (ℓ7, plus-succ, ⟨⟩) :: nil
  get_str r2, zero/0;         ℓ = ℓ4, ℓ⃗ = ⟨⟩, c = zero
                              H ← H{ℓ4 ↦ c⟨ℓ⃗⟩}
  get_val r2, r3;
  jmp r4;                     I ← C(R(r4)) = C(success)
)

success/0 ↦ code[{}] (
  succeed;
)
```

4.5 Formal Operational Semantics

The small-step operational semantics consists of three main judgements: $m \longmapsto m'$, m done, and m fails, where m fails indicates a negative result to a Prolog query, not a stuck state. There are also numerous auxilliary judgements for unification, backtracking, trail management, etc. We begin with conceptually simple cases and proceed to conceptually complex ones. The simplest instructions are mov and succeed, requiring no auxilliary judgements:

$$\frac{R(r_s) = w}{(T, S, R, \texttt{mov}\ r_d, r_s; I) \longmapsto (T, S, R\{r_d \mapsto w\}, I)}\ \text{Mov} \longmapsto \qquad \frac{}{(T, S, R, \texttt{succeed})\ \texttt{done}}\ \text{done}$$

4.5.1 Operands. The `jmp` *op* instruction takes at *operand* which allows us to jump either to a literal location or a success continuation stored in a register. The *operand evaluation* judgement $R \vdash op \Downarrow w$ resolves an operand *op* into a word w by consulting the registers R if necessary. If the operand is a code location, `jmp` simply transfers control, else if the operand is a closure, `jmp` also loads the stored environment.

$$\frac{R \vdash op \Downarrow \ell^C \quad S(\ell^C) = \texttt{code}[\Gamma]I'}{(T, S, R, \texttt{jmp}\ op; I) \longmapsto (T, S, R, I')}\ \textsc{Jmp}\text{-}\ell^C \qquad \frac{R \vdash op \Downarrow \ell^H \quad S(\ell^H) = \texttt{close}\ (w_{env}, \ell^C) \quad S(\ell^C) = \texttt{code}[\Gamma](I')}{(T, S, R, \texttt{jmp}\ op; I) \longmapsto (T, S, R\{\texttt{env} \mapsto w_{env}\}, I')}\ \textsc{Jmp}\text{-}\ell^H$$

$$\frac{}{R \vdash \ell^C \Downarrow \ell^C}\ \ell^C \Downarrow \qquad \frac{R(r) = w}{R \vdash r \Downarrow w}\ r \Downarrow$$

4.5.2 Environments. Environment tuples are constructed with the `twrite` spinal mode. This mode begins after `put_tuple` and ends when the count of remaining tuple elements reaches 0. When the spine completes, the resulting tuple is stored in the destination register specified by the initial `put_tuple`. As before, ϵ denotes an empty sequence. We also use the notation $\vec{w} :: w$ even when adding an element w to the end of a sequence $\vec{w}$. Reading tuple elements with `proj` does not require entering a spine.

$$\frac{}{(T, S, R, \texttt{put\_tuple}\ r, n; I) \longmapsto \texttt{twrite}(T, S, R, I, r, n, \epsilon)}\ \textsc{PutTuple} \longmapsto$$

$$\frac{R(r_s) = w \quad n > 0}{\texttt{twrite}(T, S, R, \texttt{set\_val}\ r_s; I, r_d, n, \vec{w}) \longmapsto \texttt{twrite}(T, S, R, I, r_d, n-1, (\vec{w} :: w))}\ \textsc{SetVal} \longmapsto$$

$$\frac{}{\texttt{twrite}(T, S, R, I, r, 0, \vec{w}) \longmapsto (T, S\{\{\ell^H \mapsto \langle \vec{w} \rangle\}\}, R\{r \mapsto \ell^H\}, I)}\ \textsc{TWrite} \longmapsto$$

$$\frac{R(r_s) = \ell^H \quad S(\ell^H) = \langle w_1, \ldots, w_i, \ldots, w_n \rangle}{(T, S, R, \texttt{proj}\ r_d, r_s, i; I) \longmapsto (T, S, R\{r_d \mapsto w_i\}, I)}\ \textsc{Proj} \longmapsto$$

4.5.3 Continuations and Backtracking. The instructions `close` and `push_bt` allocate new success and failure continuations, respectively. The `close` instruction puts the continuation in a register r_d for use by a future `jmp`, whereas `push_bt` puts the failure continuation in the trail. As shown in Section 4.5.4, the trail maintains an invariant that the location of every free variable bound since the last `push_bt` is stored in the current trail frame, which is necessary when backtracking. Because we have just created a new failure continuation, after a `push_bt` our new trail frame contains the empty list ϵ. Backtracking is handled automatically when unification fails, thus there is no need to make the failure continuation accessible via a register.

$$\frac{R(r_e) = w_{env}}{(T, S, R, \texttt{close}\ r_d, r_e, \ell^C; I) \longmapsto (T, S\{\{\ell^H \mapsto \texttt{close}\ (w_{env}, \ell^C)\}\}, R\{r \mapsto \ell^H\}, I)}\ \textsc{Close} \longmapsto$$

$$\frac{R(r_e) = w_{env}}{(T, S, R, \texttt{push\_bt}\ r_e, \ell^C; I) \longmapsto ((\epsilon, w_{env}, \ell^C) :: T, S, R, I)}\ \textsc{PushBT} \longmapsto$$

The trail invariant is essential to the correctness of the `backtrack` operation, which succeeds when there is failure continuation on the trail, or signals query failure when the trail is empty. Trail unwinding has its inverse in trail updating, which adds a recently-bound variable to the trail (or safely skips it if there are no failure continuations left).

$$\frac{\texttt{unwind}(S, t) = S' \quad C(\ell^C) = \texttt{code}[\{\texttt{env} : \tau\}](I)}{\texttt{backtrack}(S, (t, w, \ell^C) :: T) = (T, S', \{\texttt{env} \mapsto w\}, I)}\ \textsc{BT-Cons} \qquad \frac{}{\texttt{backtrack}(S, \epsilon) = \bot}\ \textsc{BT-Nil}$$

$$\texttt{unwind}(S, (\ell^H : a) :: t) = \texttt{unwind}(S\{\ell^H \mapsto \textbf{FREE}[a]\}, t) \qquad \texttt{unwind}(S, \epsilon) = S$$

$$\texttt{update\_trail}(\ell^H : a, (t, w_{env}, \ell^C) :: T) = ((\ell^H : a) :: t, w_{env}, \ell^C) :: T \qquad \texttt{update\_trail}((\ell^H : a), \epsilon) = \epsilon$$

4.5.4 Unification, Occurs Checks, and Trailing. The `get_val` instruction unifies two arbitrary Prolog terms stored at r_1 and r_2. It does so using the auxilliary judgement $\mathtt{unify}(S, T, \ell_1^H, \ell_2^H) = (S', T')$, which computes the resulting store where ℓ_1^H and ℓ_2^H are unified, or $\bot$ if unification fails. It also must compute an updated trail, because unification binds free variables, and backtracking must be able to undo those changes.

$$\frac{R(r_1) = \ell_1^H \quad R(r_2) = \ell_2^H \quad \mathtt{unify}(S, T, \ell_1^H, \ell_2^H) = (S', T')}{(T, S, R, \mathtt{get\_val}\ r_1, r_2; I) \longmapsto (T', S', R, I)}\ \textsc{GetVal} \longmapsto$$

The judgement $\mathtt{unify}(S, T, \ell_1^H, \ell_2^H) = (S', T')$ is defined by mutual recursion with the judgement $\mathtt{unify\_args}(S, T, \vec{\ell}^H, \vec{\ell}'^H) = (S', T')$ which simply unifies every ℓ_i^H with the corresponding $\ell_i'^H$. These lists $\vec{\ell}^H$ and $\vec{\ell}'^H$ correspond to the push-down list (PDL) in other presentations of the WAM. An additional judgement $\mathsf{end}(S, \ell^H)$ follows chains of **BOUND** ℓ^H pointers to their ends. Because the typing invariant for heaps ensures absence of cycles, this is guaranteed to terminate. The basic unification algorithm says to recurse if both unificands are structures, or if either is a free variable, then bind it to the other unificand. However, unification must also maintain the invariant that the heap is free of cycles, thus we employ an occurs check in our algorithm, writing $\ell_1^H \in_S \ell_2^H$ when ℓ_1^H occurs in ℓ_2^H (occurs check failure) and $\ell_1^H \notin_S \ell_2^H$ otherwise (occurs check success).[1] Additionally, we employ the `update_trail` function to maintain the trail invariants when binding free variables.

$$\frac{S(\ell_2^H) = c\langle \ell_1'^H, \ldots, \ell_n'^H\rangle \quad \ell_1^H \in_S \ell_i'^H\ (\exists i \in [n])}{\ell_1^H \in_S \ell_2^H} \in c\langle\rangle$$

$$\frac{\ell_1^H = \ell_2^H}{\ell_1^H \in_S \ell_2^H} \in= \qquad \frac{S(\ell_2^H) = \textbf{BOUND}\ \ell_2'^H \quad \ell_1^H \in_S \ell_2'^H}{\ell_1^H \in_S \ell_2^H} \in \textbf{BOUND}$$

$$\frac{\ell_1^H \neq \ell_2^H \quad S(\ell_2^H) = \textbf{FREE}[a]}{\ell_1^H \notin_S \ell_2^H} \notin \textbf{FREE} \qquad \frac{S(\ell_2^H) = \textbf{BOUND}\ \ell_2'^H \quad \ell_1^H \notin_S \ell_2'^H}{\ell_1^H \notin_S \ell_2^H} \notin \textbf{BOUND}$$

$$\frac{S(\ell_2^H) = c\langle \ell_1'^H, \ldots, \ell_n'^H\rangle \quad \ell_1^H \notin_S \ell_i'^H\ (\forall i \in [n])}{\ell_1^H \notin_S \ell_2^H} \notin c\langle\rangle$$

$$\frac{S(\ell_2^H) = \langle \ell_1'^H, \ldots, \ell_n'^H\rangle \quad \ell_1^H \in_S \ell_i'^H\ (\exists i \in [n])}{\ell_1^H \in_S \ell_2^H} \in \langle\rangle \qquad \frac{S(\ell_2^H) = \langle \ell_1'^H, \ldots, \ell_n'^H\rangle \quad \ell_1^H \notin_S \ell_i'^H\ (\forall i \in [n])}{\ell_1^H \notin_S \ell_2^H} \notin \langle\rangle$$

$$\frac{S(\ell_2^H) = \mathsf{close}\ (\ell_2'^H, \ell^C) \quad \ell_1^H \in_S \ell_2'^H}{\ell_1^H \in_S \ell_2^H} \in \mathsf{close} \qquad \frac{S(\ell_2^H) = \mathsf{close}\ (\ell_2'^H, \ell^C) \quad \ell_1^H \notin_S \ell_2'^H}{\ell_1^H \notin_S \ell_2^H} \notin \mathsf{close}$$

$$\frac{S(\ell^H) = \textbf{FREE}[a]}{\mathsf{end}(S, \ell^H) = \ell^H}\ \mathsf{end}\ \textbf{FREE} \qquad \frac{S(\ell^H) = c\langle \ell_1^H, \ldots, \ell_n^H\rangle}{\mathsf{end}(S, \ell^H) = \ell^H}\ \mathsf{end}\ c\langle\rangle$$

$$\frac{S(\ell^H) = \textbf{BOUND}\ \ell'^H \quad \mathsf{end}(S, \ell'^H) = \ell''^H}{\mathsf{end}(S, \ell^H) = \ell''^H}\ \mathsf{end}\ \textbf{BOUND}$$

$$\frac{\mathsf{end}(\ell_1^H) = \ell^H \quad \mathsf{end}(\ell_2^H) = \ell^H}{\mathtt{unify}(S, T, \ell_1^H, \ell_2^H) = (S, T)}\ \mathtt{unify} = \qquad \frac{\begin{array}{c}\mathsf{end}(S, \ell_1^H) = \ell_1'^H \quad \mathsf{end}(S, \ell_2^H) = \ell'^H_2 \quad S(\ell_2'^H) = \textbf{FREE}[a] \\ \ell_2'^H \notin_S \ell_1 \quad \mathtt{update\_trail}(T, (\ell_2'^H : a)) = T'\end{array}}{\mathtt{unify}(S, T, \ell_1^H, \ell_2^H) = (S\{\ell_2'^H \mapsto \textbf{BOUND}\ \ell_1^H\}, T')}\ \mathtt{unify}\ \textbf{FREE}$$

$$\frac{\begin{array}{c}S(\ell_1'^H) = c\langle \vec{\ell}^H\rangle \quad S(\ell_2'^H) = c\langle \vec{\ell}'^H\rangle \quad \mathsf{end}(S, \ell_1^H) = \ell_1'^H \\ \mathsf{end}(S, \ell_2^H) = \ell_2'^H \quad \mathtt{unify\_args}(S, T, \vec{\ell}^H, \vec{\ell}'^H) = (S', T')\end{array}}{\mathtt{unify}(S, T, \ell_1^H, \ell_2^H) = (S', T')}\ \mathtt{unify}\ c\langle\rangle$$

[1] Typing constraints ensure that at runtime, the occurs check is only ever invoked on Prolog terms. However, the occurs check is also of broader use in the metatheory proofs, and there it is convenient to define the occurs check on closures and tuples as well, such as in Lemma 8.

$$\frac{\mathtt{unify}(S, T, \ell_1^H, \ell_2^H) = (S', T') \quad \mathtt{unify\_args}(S', T', (\ell_2^H :: \cdots :: \ell_n^H), (\ell_2'^H :: \cdots :: \ell_n'^H)) = (S'', T'')}{\mathtt{unify\_args}(S, T, (\ell_1^H :: \cdots :: \ell_n^H), (\ell_1'^H :: \cdots :: \ell_n'^H)) = (S'', T'')}\ \text{UA-Cons} \qquad \frac{}{\mathtt{unify\_args}(S, T, \epsilon, \epsilon) = (S, T)}\ \text{UA-Nil}$$

The failure cases for unification are straightforward, but they are given here for completeness:

$$\frac{S(\ell_1'^H) = \mathbf{FREE}[a] \quad \ell_1'^H \in_S \ell_2^H \quad \mathsf{end}(S, \ell_1^H) = \ell_1'^H \quad \mathsf{end}(S, \ell_2^H) = \ell_2'^H}{\mathtt{unify}(S, T, \ell_1^H, \ell_2^H) = \bot}\ \text{U}\bot\text{1} \qquad \frac{S(\ell_2'^H) = \mathbf{FREE}[a] \quad \ell_2'^H \in_S \ell_1^H \quad \mathsf{end}(S, \ell_1^H) = \ell_1'^H \quad \mathsf{end}(S, \ell_2^H) = \ell_2'^H}{\mathtt{unify}(S, T, \ell_1^H, \ell_2^H) = \bot}\ \text{U}\bot\text{2}$$

$$\frac{S(\ell_1'^H) = c\langle \vec{w} \rangle \quad S(\ell_2'^H) = c'\langle \vec{w}' \rangle \quad c \neq c' \quad \mathsf{end}(S, \ell_1^H) = \ell_1'^H \quad \mathsf{end}(S, \ell_2^H) = \ell_2'^H}{\mathtt{unify}(S, T, \ell_1^H, \ell_2^H) = \bot}\ \text{U}\bot\text{3}$$

$$\frac{\mathtt{unify\_args}(S, T, \vec{\ell}^H, \vec{\ell}'^H) = \bot \quad S(\ell_2'^H) = c\langle \vec{\ell}'^H \rangle \quad \mathsf{end}(S, \ell_1^H) = \ell_1'^H \quad \mathsf{end}(S, \ell_2^H) = \ell_2'^H \quad S(\ell_1'^H) = c\langle \vec{\ell}^H \rangle}{\mathtt{unify}(S, T, \ell_1^H, \ell_2^H) = \bot}\ \text{U}\bot\text{4}$$

$$\frac{\mathtt{unify}(S, T, \ell^H, \ell'^H) = \bot}{\mathtt{unify\_args}(S, T, (\ell^H :: \vec{\ell}^H), (\ell'^H :: \vec{\ell}'^H)) = \bot}\ \text{UA}\bot\text{1} \qquad \frac{\mathtt{unify\_args}(S', T', \vec{\ell}^H, \vec{\ell}'^H) = \bot \quad \mathtt{unify}(S, T, \ell^H, \ell'^H) = (S', T')}{\mathtt{unify\_args}(S, T, (\ell^H :: \vec{\ell}^H), (\ell'^H :: \vec{\ell}'^H)) = \bot}\ \text{UA}\bot\text{2}$$

Lastly, if unification fails, due to the above failure rules, `get_val` tries backtracking. If the trail is non-empty, it backtracks successfully, else execution stops and the query has failed.

$$\frac{R(r_1) = \ell_1^H \quad R(r_2) = \ell_2^H \quad \mathtt{unify}(S, T, \ell_1^H, \ell_2^H) = \bot \quad \mathtt{backtrack}(S, T) = m'}{(T, S, R, \mathtt{get\_val}\ r_1, r_2; I) \longmapsto m'}\ \text{GetVal-BT}$$

$$\frac{R(r_1) = \ell_1^H \quad R(r_2) = \ell_2^H \quad \mathtt{unify}(S, T, \ell_1^H, \ell_2^H) = \bot \quad \mathtt{backtrack}(S, T) = \bot}{(T, S, R, \mathtt{get\_val}\ r_1, r_2; I)\ \mathtt{fails}}\ \text{GetVal-}\bot$$

4.5.5 Term Constructors and Occurs Checks. The `put_var` instruction immediately allocates a free variable. Structures are constructed with a write spine, initiated by `put_str`. For symmetry with `get_str`, we first allocate a free variable and replace it with a structure when the spine completes. Within a write spine, `unify_var` allocates free variables while `unify_val` copies a value into the structure. In the typical case, a write spine finishes when enough arguments have been computed (one for each constructor argument, i.e. $\mathtt{arity}(c)$) by replacing the free variable with a complete structure. However, the formalism technically allows us to refer to the destination r within its own write spine. To prevent a cycle, we perform an occurs check ($\ell^H \notin_S \ell_i^H$) and backtrack on failure. The choice to use an occurs check here was made for its resulting proof simplicity. For implementation purposes, an equally correct and more efficient choice is to enforce a syntactic restriction that prohibits references to r within its own write spine.

$$\frac{}{(T, S, R, \mathtt{put\_var}\ \ [a]r; I) \longmapsto (T, S\{\{\ell^H \mapsto \mathbf{FREE}[a]\}\}, R\{r \mapsto \ell^H\}, I)}\ \text{PutVar} \longmapsto$$

$$\frac{\Sigma(c) = \vec{a} \to a}{(T, S, R, \mathtt{put\_str}\ c, r; I) \longmapsto \mathtt{write}(T, S\{\{\ell^H \mapsto \mathbf{FREE}[a]\}\}, R\{r \mapsto \ell^H\}, I, c, \ell^H, \epsilon)}\ \text{PutStr} \longmapsto$$

$$\frac{}{\begin{array}{c}\mathtt{write}(T, S, R, \mathtt{unify\_var}\ [a]r_s; I, c, \ell_d^H, \vec{\ell}^H) \longmapsto \\ \mathtt{write}(T, S\{\{\ell^H \mapsto \mathbf{FREE}[a]\}\}, R\{r_s \mapsto \ell^H\}, I, c, \ell_d^H, (\vec{\ell}^H :: \ell^H))\end{array}}\ \text{UnifyVar} \longmapsto \text{W}$$

$$\frac{R(r_s) = \ell^H}{\mathtt{write}(T, S, R, \mathtt{unify\_val}\ r_s; I, c, \ell_d^H, \vec{\ell}^H) \longmapsto \mathtt{write}(T, S, R, I, c, \ell_d^H, (\vec{\ell}^H :: \ell^H))}\ \text{UnifyVal} \longmapsto \text{W}$$

$$\frac{S(\ell^H) = \mathbf{FREE}[a] \quad \ell^H \notin_S \ell_i^H \quad |\vec{\ell}^H| = \mathtt{arity}(c)}{\mathtt{write}(T, S, R, I, \ell^H, c, \vec{\ell}^H) \longmapsto (T, S\{\ell^H \mapsto c\langle \vec{\ell}^H \rangle\}, R, I)}\ \text{Write} \longmapsto$$

$$\frac{\ell^H \in_S \ell_i^H \quad |\vec{\ell}^H| = \mathtt{arity}(c) \quad \mathtt{backtrack}(S,T) = m'}{\mathtt{write}(T,S,R,I,\ell^H,c,\vec{\ell}^H) \longmapsto m'}\ \textsc{Write-BT}$$

$$\frac{\ell^H \in_S \ell_i^H \quad |\vec{\ell}^H| = \mathtt{arity}(c) \quad \mathtt{backtrack}(S,T) = \bot}{\mathtt{write}(T,S,R,I,\ell^H,c,\vec{\ell}^H)\ \mathtt{fails}}\ \textsc{Write-}\bot$$

4.5.6 Term Destructor `get_str`. The instruction `get_str` c, r starts a spine which unifies the content of register r with a structure $c\langle \ell_1^H, \ldots, \ell_n^H\rangle$ where each w_i is provided by the i'th spinal instruction. In the case where r contains a free variable, this amounts to building a new structure, and thus the `write` case of `get_str` simply reuses the `write` spines of `put_str`.

$$\frac{R(r) = \ell^H \quad \mathsf{end}(S,\ell^H) = \ell'^H \quad S(\ell'^H) = \mathbf{FREE}[a]}{(T,S,R,\mathtt{get\_str}\ c, r; I) \longmapsto \mathtt{write}(T,S,R,I,c,\ell'^H,\epsilon)}\ \textsc{GetStr} \longmapsto \textsc{W}$$

When r contains a structure, we perform structure-to-structure unification. This can fail in two ways: either the head constructor c does not match or one of the argument positions does not unify. The first conditon is checked during `get_str` itself, the other during the ensuing spinal instructions. In both cases, errors are handled by backtracking if possible:

$$\frac{R(r) = \ell^H \quad \mathsf{end}(S,\ell^H) = \ell'^H_H \quad S(\ell'^H) = c\langle \ell_1^H, \ldots, \ell_n^H\rangle}{(T,S,R,\mathtt{get\_str}\ c, r; I) \longmapsto \mathtt{read}(T,S,R,I,\vec{\ell}^H)}\ \textsc{GetStr} \longmapsto \textsc{R}$$

$$\frac{\begin{array}{c}R(r) = \ell^H \quad \mathsf{end}(S,\ell^H) = \ell'^H \\ S(\ell'^H) = c'\langle \ell_1, \ldots, \ell_n\rangle \quad c \neq c' \quad \mathtt{backtrack}(S,T) = m'\end{array}}{(T,S,R,\mathtt{get\_str}\ c, r; I) \longmapsto m'}\ \textsc{GetStr-BT}$$

$$\frac{\begin{array}{c}R(r) = \ell^H \quad \mathsf{end}(S,\ell^H) = \ell'^H \\ S(\ell'^H) = c'\langle \ell_1, \ldots, \ell_n\rangle \quad c \neq c' \quad \mathtt{backtrack}(S,T) = \bot\end{array}}{(T,S,R,\mathtt{get\_str}\ c, r; I)\ \mathtt{fails}}\ \textsc{GetStr-}\bot$$

$$\frac{}{\mathtt{read}(T,S,R,\mathtt{unify\_var}\ [a]r; I, (\ell^H :: \vec{\ell}^H)) \longmapsto \mathtt{read}(T,S,R\{r \mapsto \ell^H\}, I, \vec{\ell}^H)}\ \textsc{UnifyVar} \longmapsto \textsc{R}$$

$$\frac{R(r) = \ell'^H \quad \mathtt{unify}(S,T,\ell^H,\ell'^H) = (S',T')}{\mathtt{read}(T,S,R,\mathtt{unify\_val}\ r; I, (\ell^H :: \vec{\ell}^H)) \longmapsto \mathtt{read}(T',S',R,I,\vec{\ell}^H)}\ \textsc{UnifyVal} \longmapsto \textsc{R}$$

$$\frac{R(r) = \ell'^H \quad \mathtt{unify}(S,T,\ell^H,\ell'^H) = \bot \quad \mathtt{backtrack}(T) = m'}{\mathtt{read}(T,S,R,\mathtt{unify\_val}\ r; I, (\ell^H :: \vec{\ell}^H)) \longmapsto m'}\ \textsc{UnifyVal-BT}$$

$$\frac{R(r) = \ell'^H \quad \mathtt{unify}(S,T,\ell^H,\ell'^H) = \bot \quad \mathtt{backtrack}(T) = \bot}{\mathtt{read}(T,S,R,\mathtt{unify\_val}\ r; I, (\ell^H :: \vec{\ell}^H))\ \mathtt{fails}}\ \textsc{UnifyVal-}\bot$$

4.6 Metatheory

In both the SWAM and dependently-typed TWAM, the main metatheorems are progress and preservation.

Theorem (Progress). *If* $\cdot \vdash m$ ok *then either* m done *or* m `fails` *or* $m \longmapsto m'$.

Theorem (Preservation). *If* $\cdot \vdash m$ ok *and* $m \longmapsto m'$ *then* $\cdot \vdash m'$ ok.

Where m `fails` mean that a Prolog query terminated normally, but the query had no solution.

In Section 6.4, the progress and preservation results for the TWAM will be strong enough to enable certifying compilation. In the SWAM, progress and preservation amount to type and memory-safety. Because the theorem of Section 6.4 subsumes progress and preservation for SWAM, we restrict ourselves here to the commonalities and present the differences in Section 6.4. For the sake of readability, both this section and Section 6.4 give proof sketches where the reader might find a detailed proof tedious. For the sake of exhaustiveness, an extended proof for the dependent system is given in the electronic appendix, however.

The metatheory for SWAM begins with standard preliminary lemmas such as canonical forms and weakening. This is followed with the heart of the metatheory: our treatment of the occurs check and unification.

The key lemma Heap Update (Lemma 9) shows that binding free variables preserves the acyclic heap invariant when the occurs check passes, which gives us preservation for unification and thus every instruction that depends on unification.

4.6.1 *Preliminaries.*

Lemma 1 (Canonical Forms). *Canonical forms consists of a subclaim for each relevant class of values.*

- Code Values: *If* $\Psi \vdash v^C : \tau$ *then* $\tau = \neg\Gamma$ *and* $v^C = \mathsf{code}[\Gamma](I)$
- Word Values: *If* $\Psi \vdash_{\Sigma;\Xi} w : \tau$ *and* $(C, H) : (\Xi; \Psi)$ *then* w *has form* ℓ^H *or* ℓ^C *where* $\ell^H \in Dom(H)$ *or* $\ell^C \in Dom(C)$.
- Heap Values: *If* $\Psi \vdash v^H : \tau$ *then*
  - *Either* $\tau = x[\vec{\tau}]$ *and* $v = \langle \vec{w} \rangle$ *and* $\Psi \vdash \vec{w} : \vec{\tau}$
  - *Or* $\tau = a$ *and either* $v = \textbf{\textit{BOUND}}\ w$ *or* $v = \textbf{\textit{FREE}}[a]$ *or* $v = c\langle \vec{w} \rangle$ *where* $\Psi \vdash \vec{w} : \vec{a}$ *and* $\Sigma(c) = \vec{a} \to a$.
  - *Or* $\tau = \neg\Gamma$ *and* $v = \mathsf{close}\,(w, \ell^C)$ *where* $\Psi \vdash w : \neg\Gamma(\mathsf{env})$.

Proof. Each claim is by inversion on the typing rules. □

Lemma 2 (Weak Unicity of Heap Value Typing). *For any value* v*, at most one of the following holds:*

(1) $\Psi \vdash v : a$
(2) $\Psi \vdash v : x[\vec{\tau}]$
(3) $\Psi \vdash v : \neg\Gamma$

Proof. By cases on $\Psi \vdash v : \tau$. Each rule requires v to have a specific form. If $\Psi \vdash v : a$ then $v = \textbf{FREE}[a], \textbf{BOUND}\ \ell^H$, or $c\langle \ell_1^H, \ldots, \ell_n^H \rangle$. In each case the only rules that apply produce type a (the type annotation enforces unicity for $\textbf{FREE}[a]$). If $\Psi \vdash v : x[\vec{\tau}]$ then $v = \langle \vec{w} \rangle$ and the only rule that applies produces type $x[\vec{\tau}]$. Otherwise $\Psi \vdash v : \neg\Gamma$ and the rules force $v = \mathsf{code}[\Gamma]$ or $v = \mathsf{close}(w_{env}, \ell^C)$. In either case the only rule that applies produces $\Psi \vdash v : \neg\Gamma$, because the type is restricted by either the annotation Γ or by $\Xi(\ell^C)$. □

Lemma 3 (Weakening). *In SWAM we need weakening for word and heap value typing and occurs check:*

- Word Values: *If* $\Psi \vdash w : \tau$ *then* $\Psi\{\{\ell_H : \tau'\}\} \vdash w : \tau$.
- Occurs Check: *For fresh* ℓ^H*, (a) If* $\ell_1^H \in_H \ell_2^H$*, then* $\ell_1^H \in_{H\{\{\ell^H \mapsto v\}\}} \ell_2^H$ *and (b) If* $\ell_1^H \notin_H \ell_2^H$*, then* $\ell_1^H \notin_{H\{\{\ell^H \mapsto v\}\}} \ell_2^H$.
- Heap Typing: *For all fresh* ℓ^H *and even ill-typed* v^H*, if* $\cdot \vdash H : \Psi$ *then* $\cdot \vdash H\{\{\ell^H \mapsto v^H\}\} : \Psi$.

Proof. By induction on the derivation $\Psi \vdash w : \tau$, $\ell_1^H \in_H \ell_2^H$, $\ell_1^H \notin_H \ell_2^H$, or $\cdot \vdash H : \Psi$ respectively, using the fact that for fresh ℓ^H, $H\{\{\ell^H \mapsto v\}\}(\ell'^H) = H(\ell'^H)$ for all $\ell'^H \in Dom(H)$. Heap Typing weakening uses the fact that $\cdot \vdash H : \{\}$ for all H, i.e. heaps may contain inaccessible values not assigned types by Ψ. □

Lemma 4 (Register File Subtyping). *If* $\Gamma \vdash I$ ok *and* $\cdot \vdash \Gamma \leq \Gamma'$ *then* $\Gamma' \vdash I$ ok.

Proof. By induction on the derivation $\Gamma \vdash I$ ok. □

4.6.2 Occurs Check. The following theory of occurs checks is used in the theory of unification.

Lemma 5 (Occurs is a total function). *If* $\cdot \vdash H : \Psi$ *and* $\Psi \vdash_{\Sigma;\Xi} \ell_2^H : a$ *then:*

- Total: *For all* ℓ_1^H, *either* $\ell_1^H \in_H \ell_2^H$ *or* $\ell_1^H \notin_H \ell_2^H$.
- Function: *If* $\ell_1^H \notin_H \ell_2^H$ *is derivable, then* $\ell_1^H \in_H \ell_2^H$ *is not derivable.*

Proof. Totality is by induction on typing derivation of $\cdot \vdash H : \Psi$, appealing to Lemmas 3 and 1. Functionhood is by induction on the derivation $\ell_1^H \notin_H \ell_2^H$. In the cases $H(\ell_2^H) = \mathbf{FREE}[a]$, $H(\ell_2^H) = \mathbf{BOUND}\ w$, and $\texttt{close}\ (w_{env}, \ell^C)$, clearly no rules apply for $\ell_1^H \in_H \ell_2^H$. Consider the case $c\langle \ell_1^H, \ldots, \ell_n^H \rangle$ (the tuple case is symmetric):

$$\textbf{case}\ \frac{H(\ell_2^H) = c\langle \ell_1'^H, \ldots, \ell_n'^H \rangle \quad \ell_1^H \notin_H \ell_i'^H (\forall 1 \leq i \leq n)}{\ell_1^H \notin_H \ell_2^H} \notin c\langle\rangle$$

By the IH, $\forall i.(\ell_1^H \in_H \ell_i'^H$ is not derivable), so $\neg\exists i.(\ell_1^H \in_H \ell_i'^H$ is derivable). But because $H(\ell_2) = c\langle \ell_1^H, \ldots, \ell_n^H \rangle$, the only rule that might apply requires $\exists i.(\ell_1 \in_H \ell_i'$ is derivable). □

Lemma 6 (Transitivity of Occurs). *If* $\cdot \vdash H : \Psi$, $\ell_1^H \in_H \ell_2^H$, *and* $\ell_2^H \in_H \ell_3^H$, *then* $\ell_1^H \in_H \ell_3^H$.

Proof. By induction on the derivation $\ell_2^H \in_H \ell_3^H$. □

Lemma 7 (Occurs Strengthening). *If* $\cdot \vdash H : \Psi$, $\ell_2^H \in Dom(H)$, $\ell'^H \notin Dom(H)$, *and* $\ell_1^H \notin_{H\{\{\ell'^H \mapsto v'\}\}} \ell_2^H$ *then* $\ell_1^H \notin_H \ell_2^H$.

Proof. By induction on the derivation $\ell_1^H \notin_{H\{\{\ell'^H \mapsto v'\}\}} \ell_2^H$, appealing to Lemma 1. □

4.6.3 Heap Modification. The simply-typed metatheory culminates in the treatment of heap modification. We begin with a strengthening lemma:

Lemma 8 (Heap Value Strengthening). *If* $\Psi\{\{\ell_1^H : \tau_1\}\} \vdash v_2 : \tau_2$ *and* $\ell_1^H \notin_H \ell_2^H$, *then* $\Psi \vdash v_2 : \tau_2$.

Proof. By cases on $\Psi\{\{\ell_1^H : \tau_1\}\} \vdash v_2 : \tau_2$. The case $c\langle \ell_1^H, \ldots, \ell_n^H \rangle$ is representative:

$$\textbf{case}\ \frac{\Sigma(c) = \vec{a} \to \tau_2 \quad \Sigma; \Psi\{\{\ell_1^H : \tau_1\}\} \vdash \ell_i'^H : a_i}{\Psi\{\{\ell_1^H : \tau_1\}\} \vdash c\langle \ell_1^H, \ldots, \ell_n^H \rangle : \tau_2}\ \text{HV-Str}$$

Then for each $\ell_i'^H$ the typing derivation has form $\dfrac{\Psi\{\{\ell_1^H : \tau_1\}\}(\ell_i'^H) = \Im(M_i : a_i)}{\Psi\{\{\ell_1^H : \tau_1\}\} \vdash \ell_i'^H : \Im(M_i : a_i)}\ \text{WV-}\ell^H$. Note $\ell_i'^H \neq \ell_1^H$. Otherwise we would have

$$\frac{\dfrac{\ell_i'^H = \ell_1^H}{\ell_1^H \in_H \ell_i'^H (\exists i)}\ \in= \qquad H(\ell_2^H) = c\langle \ell_1'^H, \ldots \ell_n'^H \rangle}{\ell_1^H \in_H \ell_2^H} \in c\langle\rangle$$

Which would contradict $\ell_1^H \notin_H \ell_2^H$ because occurs is a function (Lemma 5). Because $\ell_1^H \neq \ell_i'^H$ and $\ell_i'^H \in Dom(\Psi\{\{\ell_1^H : \tau_1\}\})$ we have $\ell_i'^H \in Dom(\Psi)$ and

$$\frac{\Sigma(c) = a_1 \cdots \longrightarrow a_n \longrightarrow \tau_2 \qquad \dfrac{\Psi(\ell_i'^H) = \mathfrak{S}(M_i : a_i) \quad (\forall i)}{\Psi \vdash_{\Sigma;\Xi} \ell_i'^H : a_i}\ \text{WV-}\ell^H}{\Psi \vdash_{\Sigma;\Xi} c\langle \ell_1^H, \ldots, \ell_n^H \rangle : \tau_2}\ \text{HV-Str}$$

□

Lemma 9 (Heap Update). *If* $\cdot \vdash H : \Psi$ *and* $\Psi(\ell_1^H) = a$ *then*

(a) *If* $\Psi(\ell_2^H) = a$ *and* $\ell_1^H \notin_H \ell_2^H$, *(the occurs check passes)* *then* $\cdot \vdash H\{\ell_1^H \mapsto \textbf{\textit{BOUND}}\ \ell_2^H\} : \Psi$.

(b) *If for all* i, $\Psi(\ell_i'^H) = a_i$ *and* $\ell_1^H \notin_H \ell_i'^H$ *and* $\Sigma(c) = \vec{a} \longrightarrow a$, *then* $\cdot \vdash H\{\ell_1^H \mapsto c\langle \ell_1'^H, \ldots, \ell_n'^H \rangle\} : \Psi$.

The simple statement of Heap Update belies the complexity of its proof. Recall that heaps and heap types are unordered (identified up to permutation), while heap typing derivations are ordered, serving as a witness that the heap is acyclic. The proof of Heap Update must show that no cycles are introduced, which requires exhibiting a new acyclic ordering in, e.g. the derivation of $\cdot \vdash H\{\ell_1^H \mapsto$ **BOUND** $\ell_2^H\} : \Psi$.

Heap Typing Proof Terms. In the interest of rigor, we introduce proof term notation for heap typing derivations, which allows us to give a concise, explicit construction of the topological orderings required by Heap Update. The reader may wish to skip this section on a first reading, as it introduces significant proof machinery that is not needed elsewhere. Recall the typing rules for heaps:

$$\frac{}{\cdot \vdash H : \{\}}\ \text{HT-Nil} \qquad \frac{\cdot \vdash H : \Psi \quad H(\ell^H) = v^H \quad \Psi \vdash v^H : \tau \quad \ell^H \notin Dom(\Psi)}{\cdot \vdash H : \Psi\{\{\ell^H : \tau\}\}}\ \text{HT-Cons}$$

These rules result in list-structured proof terms:

$$\mathcal{D} ::= nil_H \mid \mathcal{D}; d_{\ell^H}$$

We write $\mathcal{D} : (\cdot \vdash H : \Psi)$ when $\mathcal{D}$ is a proof term of $\cdot \vdash H : \Psi$. In this notation nil_H is the proof term for HT-Nil applied to heap H and $\mathcal{D}; d_{\ell^H}$ is the proof term for HT-Cons applied to subderivations $\mathcal{D} : (\cdot \vdash H : \Psi)$ and $d_{\ell^H} : (\Psi \vdash v^H : \tau)$ and it is a proof of $\cdot \vdash H : \Psi\{\{\ell^H : \tau\}\}$. To state the key lemma precisely, we exploit several functions over proof terms.

The notation $\texttt{pred}(\mathcal{D}, \ell^H)$ denotes the set of heap locations assigned types by $\mathcal{D}$ that appear before ℓ^H within $\mathcal{D}$ and $\texttt{succ}(\ell^H)$ denotes the set of locations that appear after it. The notation $\texttt{elems}(\mathcal{D})$ denotes all locations assigned types by $\mathcal{D}$. They can be defined recursively by:

$$\texttt{elems}(nil_H) = \emptyset \quad \texttt{elems}(\mathcal{D}; d_{\ell^H}) = \{\ell^H\} \cup \texttt{elems}(\mathcal{D})$$

$$\texttt{pred}(nil_H, \ell^H) = \emptyset \qquad \texttt{pred}((\mathcal{D}; d_{\ell^H}), \ell^H) = \texttt{elems}(\mathcal{D}) \qquad \texttt{pred}((\mathcal{D}; d_{\ell'^H}), \ell^H) = \texttt{pred}(\mathcal{D}, \ell^H)$$

$$\texttt{succ}(nil_H, \ell^H) = \emptyset \qquad \texttt{succ}((\mathcal{D}; d_{\ell^H}), \ell^H) = \emptyset \quad \texttt{succ}((\mathcal{D}; d_{\ell'^H}), \ell^H) = \texttt{succ}(\mathcal{D}, \ell^H) \cup \{\ell'^H\}$$

We now have the machinery to state a subclaim which entails both claims of Heap Update.

subclaim 1 (Heap Reordering). *If* $\mathcal{D} : (\cdot \vdash H : \Psi), H(\ell_1^H) =$ ***FREE***$[a], \ell_1^H \in Dom(\Psi), \ell_2^H \in Dom(\Psi), \ell_1^H \notin_H \ell_2^H$ *then exists* $\mathcal{D}' : (\cdot \vdash H : \Psi)$ *where* $\texttt{succ}(\mathcal{D}', \ell_1^H) \subseteq \texttt{succ}(\mathcal{D}, \ell_1^H)$ *and* $\texttt{pred}(\mathcal{D}', \ell_2^H) \subseteq \texttt{pred}(\mathcal{D}, \ell_2^H)$ *and* $\ell_2^H \in \texttt{pred}(\mathcal{D}', \ell_1^H)$

Proof. By lexicographic induction on $|\mathcal{D}|$ and $|\texttt{succ}(\mathcal{D}, \ell_1^H)|$. We give an explicit construction of the proof term $\mathcal{D}'$ as a function of $\mathcal{D}, \ell_1^H$, and ℓ_2^H in functional pseudocode, then show the construction obeys the desired properties in each case. Here $str(d_{\ell^H})$ and $weak(d_{\ell^H})$ refer to the

typing derivations that result from appeals to the heap value strengthening and weakening lemmas, respectively (Lemmas 8 and 3).

Case	$\mathcal{D}'(\mathcal{D}, \ell_1, \ell_2) =$
	$\texttt{case}\ \mathcal{D}\ \texttt{of}$
1	$\quad \texttt{nil} \Rightarrow \mathcal{D}$
2	$\quad \mid \texttt{nil}', d_\ell \Rightarrow \mathcal{D}$
3	$\quad \mid \mathcal{D}_1; d_\ell; d_{\ell 1} \Rightarrow \mathcal{D}$
4	$\quad \mid \mathcal{D}_1; d_{\ell 1}; d_{\ell 2} \Rightarrow \mathcal{D}_1; \texttt{str}(\mathsf{d}_{\ell 2}); \texttt{weak}(\mathsf{d}_{\ell 1})$
5	$\quad \mid \mathcal{D}_1; d_\ell; d_{\ell 2}\ \texttt{where}\ \ell \neq \ell_1 \Rightarrow$
	$\quad\ \texttt{if}\ (\ell_1' \notin \ell_2)\ \texttt{then}$
5a	$\quad\quad \mathcal{D}'((\mathcal{D}_1; \texttt{str}(\mathsf{d}_{\ell 2})), \ell_1, \ell_2); \texttt{weak}(\mathsf{d}_\ell)$
	$\quad\ \texttt{else}$
5b	$\quad\quad \texttt{let}\ \mathcal{D}_2 = \mathcal{D}'((\mathcal{D}_1; d_\ell), \ell_1, \ell)\ \texttt{in}$
	$\quad\quad \mathcal{D}'((\mathcal{D}_2; d_{\ell 2}), \ell_1, \ell_2)$
	$\quad \mid \mathcal{D}_1; d_\ell; d_{\ell'}\ \texttt{where}\ \ell' \neq \ell_1, \ell' \neq \ell_2 \Rightarrow$
	$\quad\ \mathcal{D}'((\mathcal{D}_1; d_\ell), \ell_1, \ell_2); d_{\ell'}$

- Cases 1 and 2 hold vacuously because our preconditions only hold for $|\mathcal{D}| \geq 2$.
- Case 3: In this case $\ell_2^H \in \mathsf{pred}(\mathcal{D}, \ell_1^H)$ (either $\ell'^H = \ell_2^H$ or ℓ_2^H appears elsewhere in H), so there is no work to be done.
- Case 4: By the assumption that $\ell_1^H \notin_H \ell_2^H$, we can apply Lemma 8, yielding $\cdot \vdash H'\{\{\ell_2^H \mapsto v_2\}\} : \Psi'\{\{\ell_2^H : \tau_2\}\}$. By Lemma 3, $\Psi'\{\{\ell_1^H : v_1\}\} \vdash v_1 : \tau_1$, so $\cdot \vdash H'\{\{\ell_2^H \mapsto v_2, \ell_1^H \mapsto v_1\}\} : \Psi'\{\{\ell_2^H : \tau_2, \ell_1^H : \tau_1\}\}$, which satisfies the requirements.
- Case 5: By Lemma 5, either $\ell'^H \in_H \ell_2^H$ or $\ell'^H \notin_H \ell_2^H$.
- Case 5a: By Lemma 8, $H\{\{\ell_2^H \mapsto v_2\}\} : \Psi\{\{\ell_2^H \mapsto \tau_2\}\}$ so we can apply the IH on $H'\{\{\ell_2^H \mapsto v_2\}\}$ giving a derivation $\mathcal{D}_1$. The result follows in combination with Lemma 3 on ℓ^H.
- Case 5b: By Lemma 5, $\ell_1^H \in_H \ell'^H$ or $\ell_1^H \notin_H \ell'^H$. In this case $\ell_1^H \notin_H \ell'^H$. Otherwise by Lemma 6 $\ell_1^H \in_H \ell_2^H$, but we assumed $\ell_1^H \notin_H \ell_2^H$ and this is a contradition because occurs is a function (Lemma 5). Thus we can apply the IH on $H'\{\{\ell'^H \mapsto v'\}\}$ (because $|H'\{\{\ell'^H \mapsto v'\}\}| < |H|$) to swap ℓ_1^H with ℓ'^H resulting in a derivation $\mathcal{D}_2$. The IH tells us $|\mathsf{succ}(\mathcal{D}_2, \ell_1^H)| < |\mathsf{succ}(\mathcal{D}, \ell_1^H)|$, allowing us to apply the IH a second time on the derivation $\mathcal{D}_2; d_2$. The second IH implies the result.
- Case 6: This case is direct by the IH.

□

Given Heap Reordering, the first claim of Heap Update follows directly with the following derivation for **BOUND** ℓ_2^H

$$\dfrac{\dfrac{\Psi'(\ell_2^H) = \tau}{\Psi' \vdash \ell_2^H : \tau}\ \text{WV-}\ell^H}{\Psi' \vdash \textbf{BOUND}\ \ell_2^H : \tau}\ \text{HV-Bound}$$

where Ψ' is the heap type assigned by $\mathcal{D}'$ to the prefix of ℓ_2^H, which must contain ℓ_1^H.

The second claim follows by iterating Heap Reordering, and because Heap Reordering preserves the predecessors of ℓ_1^H.

4.6.4 *Trails.*

Lemma 10 (Trail Update). *Introducing and binding free variables both preserve the validity of the trail:*

(a)If $S \vdash T$ ok *and* $S(\ell^H) = \textbf{\textit{FREE}}[a]$ *then* $S\{\ell^H \mapsto w\} \vdash \texttt{update\_trail}(\ell^H : a, T)$ ok.

(b)If $S \vdash T$ ok *and* ℓ^H *is fresh then* $S\{\{\ell^H \mapsto \textbf{\textit{FREE}}[a]\}\} \vdash T$ ok.

First consider the typing rule for nonempty trails:

$$\frac{\cdot \vdash S' : (\Xi, \Psi') \quad \Psi' \vdash w_{env} : \tau \quad \texttt{unwind}(S, t) = S' \quad S' \vdash T' \text{ ok} \quad \Psi' \vdash \ell^C : \neg\{\texttt{env} : \tau\}}{S \vdash (t, w_{env}, \ell^C) :: T' \text{ ok}} \text{ Trail-Cons}$$

Trails are well-typed so long as unwinding (as used in backtracking) results in a well-typed state.

Claim (a) says binding a free variable X to a term represented by word value w, then adding X to the trail, results in a well-typed trail. $S(\ell^H) = \textbf{FREE}[a]$ then $S\{\ell^H \mapsto w\} \vdash \texttt{update\_trail}(\ell^H : a, T)$ ok iff unwinding results in a well-typed store. Because it unwinds to the same store as does T, this is true by assumption.

Claim (b) is weakening principle for trails, which follows from weakening for stores. This claim shows that the trail does not need to be modified when a fresh variable is allocated, only when it is bound to a term. Claim (b) follows from the following subclaim:

subclaim 2. $\texttt{unwind}(S\{\{\ell^H \mapsto \textbf{\textit{FREE}}[a]\}\}, t) = (S'\{\{\ell^H \mapsto \textbf{\textit{FREE}}[a]\}\})$

The subclaim holds by induction on t, completing the proof of Lemma 10.

Lemma 11 (Backtracking Totality). *For all trails* T, *if* $\cdot \vdash S : (\Xi; \Psi)$ *and* $S \vdash_{\Sigma;\Xi} T$ ok *then either* $\texttt{backtrack}(S, T) = m'$ *and* $\cdot \vdash m'$ ok *or* $\texttt{backtrack}(S, T) = \bot$

Proof. By cases on $S \vdash T$ ok. □

4.6.5 Dynamic Unification. Unification uses a simple lemma on pointer following:

Lemma 12 (End Correctness). *If* $\Psi \vdash \ell^H : a$ *and* $\cdot \vdash H : \Psi$ *then* $\texttt{end}(H, \ell^H) = \ell'_H$ *and* $\Psi \vdash \ell'^H : a$ *and either* $H(\ell'^H) = \textbf{\textit{FREE}}[a]$ *or* $H(\ell'^H) = c\langle \vec{w} \rangle$

Proof. By induction on the derivation $\Psi \vdash \ell^H : a$. □

Runtime unification is total and results in a well-typed store and trail.

Lemma 13 (Soundness of unify). *If* $\cdot \vdash H : \Psi, \Psi \vdash w_1 : a, \Psi \vdash w_2 : a$, *then* $\texttt{unify}(S, T, w_1, w_2) = \bot$ *or* $\texttt{unify}(S, T, w_1, w_2) = (S', T')$ *where* $\cdot \vdash S' : (\Psi; \Xi)$ *and* $S' \vdash T'$ ok

Proof. We prove the claim by simultaneous induction with the following subclaim:

claim 1. *For all argument lists (push-down lists)* $(w_1 :: \cdots :: w_n)$ *and* $, (w'_1 :: \cdots :: w'_n)$, *if for all* i, $\Psi \vdash w_i : a_i$ *and* $\Psi \vdash w'_i : a_i$ *then* $\texttt{unify\_args}(S, T, (w_1 :: \cdots :: w_n), (w'_1 :: \cdots :: w'_n), \ldots, (w_n, w'_n)) = \bot$ *or*
$\texttt{unify\_args}(S, T, (w_1 :: \cdots :: w_n), (w'_1 :: \cdots :: w'_n)) = (S', T')$ *where* $\cdot \vdash S' : (\Xi; \Psi)$ *and* $S' \vdash T'$ ok.

Proof. Lemma 13 is by induction on the size of the type Ψ and Claim 1 is by induction on the argument lists $(w_1 :: \cdots :: w_n), (w'_1 :: \cdots :: w'_n)$. The inductive case proceeds by cases on the form of $\texttt{end}(S, \ell^H_1)$ and $\texttt{end}(S, \ell^H_2)$ using Lemma 12. We present only a few of the success cases here, for the remaining cases are straightforward.
case $(\textbf{FREE}[a], \textbf{FREE}[a])$:

Case on $\texttt{end}(S, \ell^H_1) = \texttt{end}(S, \ell^H_2)$ holds. If it does, we apply the first rule, else by case assumption we have $\texttt{end}(S, \ell^H_2) \notin_S \ell^H_1$ and apply the second, then apply Lemma 9 to get $S\{\ell^H_1 \mapsto \textbf{BOUND}\ \ell^H_2\} : (\Psi; \Xi)$ and Lemma 10 to get $S\{\ell^H_1 \mapsto \textbf{BOUND}\ \ell^H_2\} \vdash T'$ ok. :

$$\frac{\mathsf{end}(\ell_1^H) = \ell^H \quad \mathsf{end}(\ell_2^H) = \ell^H}{\mathtt{unify}(S, T, \ell_1^H, \ell_2^H) = (S, T)}\ \mathtt{unify} = \frac{\begin{array}{c}\mathsf{end}(S, \ell_1^H) = \ell_1'^H \quad \mathsf{end}(S, \ell_2^H) = \ell_2'^H \quad S(\ell_2'^H) = \mathbf{FREE}[a] \\ \Psi(\ell_1'^H) = a \quad \ell_2'^H \notin_S \ell_1^H \quad \mathtt{update\_trail}(T, (\ell_2'^H : a)) = T'\end{array}}{\mathtt{unify}(S, T, \ell_1^H, \ell_2^H) = (S\{\ell_2'^H \mapsto \mathbf{BOUND}\ \ell_1^H\}, T')}\ \mathtt{unify}\ \mathbf{FREE}$$

case $(c\langle \ell_1^H, \ldots, \ell_n^H\rangle, c\langle \ell_1'^H, \ldots, \ell_n'^H\rangle)$

We apply the IH on the subclaim for `unify_args` and the result follows immediately (and similarly if `unify_args` were to fail).

$$\frac{\begin{array}{c}S(\ell_1'^H) = c\langle \vec{\ell}^H\rangle \quad S(\ell_2'^H) = c\langle \vec{\ell}'^H\rangle \quad \mathsf{end}(S, \ell_1^H) = \ell_1'^H \\ \mathsf{end}(S, \ell_2^H) = \ell_2'^H \quad \mathtt{unify\_args}(S, T, \vec{\ell}^H, \vec{\ell}'^H) = (S', T')\end{array}}{\mathtt{unify}(S, T, \ell_1^H, \ell_2^H) = (S', T')}\ \mathtt{unify}\ c\langle\rangle$$

□

□

The essential cases of progress and preservation for the simply-typed system are instructions such as `get_val` and `unify_val` that rely on unification. Those cases of progress and preservation follow from the unification soundness lemma above. Moreover, progress and preservation for the simply-typed system are subsumed by their dependently-typed equivalents. Most of the proofs above carry over readily. Any important differences are covered in Section 6.4.

5 TYPED COMPILATION IN PROOF-PASSING STYLE

Our certification approach is based on specifying the semantics of a T-Prolog program as an LF signature. Before we can certify the correctness of compilation, we give a mechanical translation from T-Prolog programs to LF signatures (see Figure 5.1 for example):

- A type a in T-Prolog translates to an LF constant a: `type`.
- A constructor c: $\vec{a} \rightarrow a$ translates to an LF constant of the same type.
- A predicate p: $\vec{a} \rightarrow$ `prop` translates to an LF constant p: $\vec{a} \rightarrow$ `type`.
- A clause C of form G `:-` $\mathrm{SG}_1, \ldots, \mathrm{SG}_n$. translates to an LF constant (of dependent function type) C: $\Pi\ \Delta.\Pi\ \vec{SG}.G$ where Δ consists of the free variables of $\Pi\ \vec{SG}.G$.
- Executing a query `?-`G. translates to searching for a proof of G.

Example 5.1 (T-Prolog Program with its LF Signature).

```
nat:type.                          nat:type.
zero:nat.                          zero:nat.
succ:nat → nat.                    succ:nat → nat.
plus:nat → nat → nat → prop.       Plus:nat → nat → nat → type.
plus(zero,X,X).                    Plus-Z:ΠX:nat. plus zero X X.
plus(succ(X),Y,succ(Z)) :-         Plus-S:ΠX:nat. ΠY:nat. ΠZ:nat.
  plus(X,Y,Z).                       ΠD:plus X Y Z.
                                       plus (succ X) Y (succ Z).
```

Now that we have defined the semantics of a Prolog program in LF, we can describe our certification approach. The TWAM certification approach can be summed up in the following slogan:

Proof-Carrying Code + Programming As Proof Search = Proof-Passing Style

Proof-carrying code is the technique of packaging compiled code with a formal proof that the code satisfies some property. Previous work [18] has used proof-carrying code to build certifying compilers which produce proofs that the programs they output are memory-safe. Our insight is

that by combining this technique with the programming-as-proof-search paradigm that underlies logic programming, our compiler can produce proofs of a much stronger property: partial dynamic correctness.

The programming-as-proof-search paradigm tells us that partial dynamic correctness consists of the following theorem, stated informally here and formally in Section 6.4:

Theorem 1: If a query ?-G. succeeds, there exists a proof M of G in LF.

Our compiler need only output enough information that the TWAM typechecker can reconstruct the proof of Theorem 1. This requires statically proving that whenever *any* proof search procedure p would return, the corresponding predicate P would have a proof in LF. This proof boils down to accumulating facts when unification succeeds and annotating all return points with the resultant LF proof terms. This *proof-passing* style of programming is essential to the type system of the TWAM. It is worth noting also that proof-passing style is needed only at compile-time, because TWAM also supports *proof-erasure*. Thus the only runtime cost of certifying compilation with TWAM is the cost of using SWAM vs. other variants of the WAM; TWAM introduces no additional overheads compared to SWAM.

6 DEPENDENTLY-TYPED WAM

The simply-typed system presented in Section 4 is insufficient to prove that compiled programs implement a given LF signature. The `succeed` instruction

$$\frac{}{\Gamma \vdash \texttt{succeed}; I\ \texttt{ok}}$$

trivially typechecks in any context, but we wish to prove that a program only succeeds if a proof M of some query A exists in LF. We begin our dependently-typed development by requiring exactly that in the typing rule:

$$\frac{\Delta \vdash M : A}{\Delta; \Gamma \vdash \texttt{succeed}[M : A]; I\ \texttt{ok}}$$

Yet if this was the only change we made, we could never compile meaningful programs because the premise would be too difficult to fulfill. We make this premise easier to meet by introducing the ability for continuations to accept LF proof terms as arguments. Because the `succeed[M:A]` instruction generally occurs in the top-level success continuation for a query, we can make this continuation accept a proof of M as an argument x and supply x as the proof term for `succeed`, passing the burden of proof onto the caller.

In this way, we can decompose the proof argument for M into one argument for each basic block of the proof search algorithm. This too is nontrivial: whether the proof M exists for a given query A cannot be known until A is executed at runtime, but certification occurs at compile-time. In order to reason statically about runtime proof search, the type system must connect LF terms with runtime constructs such as registers and heap values. Whenever a unification succeeds at runtime (i.e. we learn that we can apply a particular rule), we need some way to say the same terms should be unified in the statics. Without a mechanism for translating between the runtime values and the static LF terms, we have no mechanism by which to learn new facts during proof search, and thus no way to construct nontrivial LF proofs.

The simplest possible relation between an LF term M and a heap value v is the notion of equality. Since heap values can also contain pointers and can exhibit sharing structures not visible in the LF term, we might more accurately think of this equality relation as "v encodes M". We add this notion to our type system by introducing *singleton types* $\Im(M : a)$ for values that encode an LF term M which itself has type a. This is the only fundamentally new value type, though other aspects of the type system will change as well to accommodate the presence of proofs.

In particular, we introduce a context Δ that contains the types of all LF variables in scope, which are introduced either in the parameters of a code value or by the `put_var` instruction. Furthermore, we introduce a notion of static unification $M_1 \sqcap M_2$ which allows us to import knowledge learned from runtime unification into an LF proof.

To see the interaction between runtime and static unification concretely, consider the zero case of `plus`, which (using the proof terms of Example 5.1) compiles to

Example 6.1 (TWAM Compilation)**.**

```
plus-zero/3 ↦ code[Π X, Y, Z : nat.{r1 : Ϭ(X), r2 : Ϭ(Y), r3 : Ϭ(Z), r4 : Π_ : (Plus X Y Z). ¬{}}](
  put_tuple r4, 4;
    set_val r1;
    set_val r2;
    set_val r3;
    set_val ret;
 # plus-succ/3, not shown, has the same three natural number
 # parameters X, Y, Z, so we pass them in when constructing
 # the failure continuation
  push_bt r4, (plus-succ/3 X Y Z);

  get_str r1, zero/0;
  get_var r2, r3;
  jmp (ret (Plus-Z Y));
)
```

The syntax `code`$[\Delta.\Gamma](I)$ denotes a code value with body I which expects the register file to have type Γ and where Γ may refer to the LF variables in Δ. The line `jmp (r4 (Plus-Z Y))` is an example of proof passing in action. Here the success continuation `r4` expects a proof of the relevant predicate: in this case `plus X Y Z`. The `jmp` instruction constructs a proof `Plus-Z Y` to satisfy this requirement. The proof `Plus-Z Y` has type `plus zero Y Y`, so this code only typechecks if $X = zero$ and $Y = Z$, which is exactly what we learn when `get_str` and `get_var` succeed, respectively.

6.1 Instruction Statics and Dynamics

In this section we detail the type system changes made to support LF terms and singleton types. The instruction set and dynamic judgements are fundamentally identical to that of the SWAM, but both are augmented with additional annotations as needed by the addition of LF. For example, because the `put_var` instruction introduces an LF variable x, we now write `put_var` $r, x : a.\ I$ instead of `put_var` $[a]r; I$ to indicate that there is an LF variable $x : a$ in scope for the remaining instructions I, a feature we will use to write proof terms. Complete dynamics for the dependent TWAM are given in the electronic appendix. In the following sections we detail the judgements that differ significantly from the simply-typed system. A listing is given in Table 1.

6.1.1 LF Terms. The typing rule for `succeed` is as given in Section 6:

$$\frac{\Delta \vdash M : A}{\Delta; \Gamma \vdash \mathsf{succeed}[M : A]; I\ \mathsf{ok}}\ \textsc{Succeed}$$

Here M is an LF term and A is an LF type (we write A for arbitrary LF type families and a for types corresponding specifically to Prolog terms). Thus we extend the syntax of TWAM with the syntax of LF (here c stands for type family constants and term constants):

Judgement(s)	Meaning	Defined In
$\Delta;\Gamma \vdash_{\Sigma;\Xi} I$ ok	Basic Block Well-Typed	6.1
$\Delta;\Gamma \vdash I :_{s\,\Sigma;\Xi} J, I :_{t\,\Sigma;\Xi} J$	Spine Well-Typed	6.1.5, 6.1.6
$\Delta \vdash M_1 \sqcap M_2 = \sigma, \bot$	Static Unification	6.1.4
$\Delta \vdash M_1 \in M_2, M_1 \notin M_2$	Static Occurs Check	6.1.4
$\Delta;\Gamma \vdash op : \tau$	Operand Well-Typed	6.1.2
$\Delta \vdash M : A$	LF Term Well-Typed	[10]
$\Delta \vdash A : K$	LF Type Family Well-Kinded	[10]
$\Delta;\Gamma \vdash v : \tau$	Heap Value Well-Typed	6.2
$\Delta \vdash H : \Psi$	Heap File Well-Typed	4.3
$\Delta;\Gamma \vdash w : \tau$	Word Value Well-Typed	6.1.2
$\Delta;\Psi \vdash R : \Gamma$	Register File Well-Typed	6.3
$\cdot \vdash (\Delta;\mu) : H$	Heap Mapping Unique	6.3
$\Delta \vdash T$ ok	Trail Well-Typed	6.3
$\Delta;\Psi \vdash \vec{\ell}^H$ reads J_s	Prolog Read Spine Invariant	6.3
$\Delta;\Psi \vdash (\vec{\ell}^H, \ell^H, c)$ writes J_s	Prolog Write Spine Invariant	6.3
$\Delta;\Psi \vdash (n, r, \vec{\ell}^H)$ writes J_t	Tuple Spine Invariant	6.3
$\cdot \vdash m$ ok	Machine Well-Typed	6.3
$R \vdash op \Downarrow w$	Operand Evaluation	6.1.2
$R \vdash op \rightsquigarrow w$	Operand Resolution	6.1.2
$w \Downarrow w'$	Word Evaluation	6.1.2
w path, w canon	Word Canonical Forms	6.1.2
$\mathsf{end}(S, \ell^H)$	Pointer Following	4.5.4
$\ell_1^H \in_S \ell_2^H$	Dynamic Occurs Check	4.5.4
$\mathsf{unify}(S, T, \ell_1^H, \ell_2^H)$	Dynamic Unification	4.5.4
$\mathsf{unify\_args}(S, T, \vec{\ell}^H, \vec{\ell}'^H)$	Dynamic Unification	4.5.4
$\mathsf{update\_trail}(x@\ell^H : a, T) = T'$	Trail Update	6.3
$\mathsf{unwind}(S, \Delta, t) = (\Delta, S)$	Trail Unwinding	6.3
$\mathsf{backtrack}(S, T) = m, \bot$	Backtracking	4.5.3
$m \longmapsto m', m$ fails, m done	Stepping	4.5

Table 2. Index of Typing and Evaluation Judgements

LF Kinds	$K ::= \mathtt{type} \mid \Pi x : A.K$
LF Type Families	$A ::= c \mid \Pi x : A.A \mid A\ M$
LF Terms	$M ::= x \mid c \mid M\ M \mid \Pi x : A.M$

Note that the TWAM need not be instrumented with LF proof terms at runtime: LF proofs are merely given as type annotations as an aid to establishing the metatheorem of Section 6.4. LF terms make numerous appearances in TWAM. For example, because Putvar introduces a free variable at runtime, it introduces an LF variable $x : a$ in the statics as well:

$$\frac{\Delta, x : a; \Gamma\{r : \Xi(x : a)\} \vdash I \text{ ok}}{\Delta;\Gamma \vdash \mathtt{put\_var}\ r, x : a.\ I \text{ ok}}\ \textsc{Putvar}$$

6.1.2 Words and Operands. The typing rules Jmp and Mov appear as before:

$$\frac{\Xi(\ell^C) = \neg\Gamma' \quad \cdot \vdash \Gamma' \leq \Gamma}{\Delta;\Gamma \vdash_{\Sigma;\Xi} \mathtt{jmp}\ op, I \text{ ok}}\ \textsc{Jmp} \qquad \frac{\Delta;\Gamma \vdash op : \tau \quad \Delta;\Gamma\{r_d : \tau\} \vdash I \text{ ok}}{\Delta;\Gamma \vdash \mathtt{mov}\ r_d, op; I \text{ ok}}\ \textsc{Mov}$$

However, both instructions rely on *operands*. In TWAM, we generalize operands (and word values) so they can accept LF terms as arguments:

$$\begin{array}{rl} \text{operands} & op ::= \ell^C \mid r \mid op\ M \mid \lambda x : A.\ op \\ \text{word values} & w ::= \ell^C \mid \ell^H \mid w\ M \mid \lambda x : A.\ w \end{array}$$

With this change we also update the statics and (big-step) dynamics for operands and word values. The main dynamic judgement is still $R \vdash op \Downarrow w$ (Operand Evaluation), but we add auxilliary judgements $R \vdash op \rightsquigarrow w$ (Operand Resolution) and $w \Downarrow w'$ (Word Evaluation).

$$\frac{}{\ell^C \Downarrow \ell^C}\ \ell^C \Downarrow \qquad \frac{w \Downarrow w'}{(\lambda x : A.\ w) \Downarrow (\lambda x : A.\ w')}\ \lambda \Downarrow \qquad \frac{op \Downarrow (\lambda x : A.\ w') \quad [M/x]w' \Downarrow w''}{op\ M \Downarrow w''}\ \beta \Downarrow \qquad \frac{op \Downarrow w \quad w\ \texttt{path}}{op\ M \Downarrow w\ M}\ op\ M \Downarrow$$

$$\frac{R \vdash op \rightsquigarrow w}{R \vdash op\ M \rightsquigarrow w\ M}\ op\ M \rightsquigarrow \qquad \frac{R \vdash op \rightsquigarrow w}{R \vdash (\lambda x : A.\ op) \rightsquigarrow (\lambda x : A.\ w)}\ \lambda \rightsquigarrow \qquad \frac{R(r) = w}{R \vdash r \rightsquigarrow w}\ r \rightsquigarrow \qquad \frac{R \vdash op \rightsquigarrow w \quad w \Downarrow w'}{R \vdash op \Downarrow w'}\ op \Downarrow$$

$$\frac{\Gamma(r) = \tau}{\Delta; \Gamma \vdash r : \tau}\ op\ r \qquad \frac{\Delta; \Gamma \vdash w : \Pi x : A.\ \tau \quad \Delta \vdash M : A}{\Delta; \Gamma \vdash (w\ M) : [M/x]\tau}\ op\ (w\ M) \qquad \frac{\Delta \vdash A : \mathsf{type} \quad \Delta, x : A; \Gamma \vdash w : \tau}{\Delta; \Gamma \vdash (\lambda x : A.\ w) : (\Pi x : A.\ \tau)}\ op\ \lambda$$

$$\frac{\Psi(\ell^H) = \tau}{\Delta; \Psi \vdash \ell^H : \tau}\ w\ \ell^H \qquad \frac{\Delta; \Psi \vdash w : \Pi x : A.\ \tau \quad \Delta \vdash M : A}{\Delta; \Psi \vdash (w\ M) : [M/x]\tau}\ w\ (w\ M) \qquad \frac{\Delta \vdash A : \mathsf{type} \quad \Delta, x : A; \Psi \vdash w : \tau}{\Delta; \Psi \vdash (\lambda x : A.\ w) : (\Pi x : A.\ \tau)}\ w\ \lambda$$

In Jmp, the generalization of operands supports proof-passing. In Mov, it supports tail-call optimization as used in Section 6.5. As in LF, we have a notion of canonical forms for words, written $w\ \texttt{canon}$, with an auxilliary judgement $w\ \texttt{path}$:

$$\frac{}{\ell\ \texttt{path}}\ \ell\ \texttt{path} \qquad \frac{w\ \texttt{path}}{w\ M\ \texttt{path}}\ w\ M\ \texttt{path} \qquad \frac{w\ \texttt{path}}{w\ \texttt{canon}}\ w\ \texttt{canon} \qquad \frac{w\ \texttt{canon}}{(\lambda x : A.\ w)\ \texttt{canon}}\ \lambda\ \texttt{canon}$$

To simplify the proofs, the typing invariants for machine states require canonicity. However, because canonical forms always exist [10] and involve only static-level computation, the choice of when to require canonical forms is irrelevant.

6.1.3 Continuations. The rules Close and BT also use operands to track LF proof terms in closures, but those operands are syntactically restricted to $\ell^C\ \vec{M}$ in order to avoid closures within closures, which would needlessly complicate the dynamics. Furthermore, we see in Close that the type of continuations has been generalized to $\Pi\vec{x} : \vec{A}.\ \neg\Gamma'$: a continuation can take any number of LF terms, which may freely mix Prolog terms and proof terms. Here the terms $\vec{M}$ are a static component of the environment, stored in the closure, while the $\vec{x}$ are static arguments supplied by the caller:

$$\frac{\Gamma(r_s) = \tau \quad \Delta; \Gamma\{r_d : \Pi\vec{x} : \vec{A}.\ \neg\Gamma'\} \vdash I\ \mathsf{ok}}{\Delta; \Gamma \vdash (\ell^C\ \vec{M}) : \Pi\vec{x} : \vec{A}.\ \neg\Gamma'\{\mathsf{env} : \tau\}}\ \textsc{Close} \qquad \frac{\Delta; \Gamma \vdash I\ \mathsf{ok} \quad \Gamma(r) = \tau \quad \Delta; \Gamma \vdash (\ell^C\ \vec{M}) : \neg\{\mathsf{env} : \tau\}}{\Delta; \Gamma \vdash \mathtt{push\_bt}\ r, (\ell^C\ \vec{M}); I\ \mathsf{ok}}\ \text{BT}$$

$$\Delta; \Gamma \vdash \mathtt{close}\ r_d, r_s, (\ell^C\ \vec{M}); I\ \mathsf{ok}$$

6.1.4 Static Unification. We arrive now at what is arguably the most novel and surprising technical result of the TWAM type system: Static unification as used in the TWAM type system is not only in harmony with Prolog-style runtime unification, but is strong enough to enable the type-checking of LF proofs. Without our static unification mechanism, it would in general be impossible to show the proof-terms returned by a clause were well-typed (consider the `jmp` in Example 6.1).

As before, `get_val` unifies two Prolog terms stored in registers r_1, r_2. Thanks to the addition of singleton types, the type system now has access to LF terms $M_1, M_2 : a$ describing the values of r_1, r_2. The subtlety of static unification lies in the fact that because the exact values of r_1 and r_2 are unknown until runtime, the terms M_1 and M_2 cannot be the exact values of r_1 and r_2. Rather, they will merely be some terms that *unify* with the eventual values of r_1 and r_2. What we find novel

and surprising is that this partial knowledge represented by M_1 and M_2 is simultaneously strong enough to certify proof search, yet consistent with the actual behavior at runtime.

To typecheck `get_val`, we unify the terms M_1, M_2 at compile-time. We write $\Delta \vdash M_1 \sqcap M_2 = \sigma$ to say they successfully unify with most-general unifier σ. We apply the substitution σ while typechecking the remaining instructions I. The substitution notation $[[\sigma]]\Delta$ indicates that σ substitutes for an arbitrary set of variables from Δ (i.e. $Dom(\sigma) \subseteq Dom(\Delta)$) and that the replacees $Dom(\sigma)$ should be removed from Δ in the process: the need for this variant of substitution arises because the variables of a most-general unifier $Dom(\sigma)$ may appear at arbitrary positions throughout Δ.

The rule Getval-F says it is also possible that we statically detect unification failure, written $\Delta \vdash M_1 \sqcap M_2 = \bot$, in which case the program is vacuously well-typed because unification will certainly fail at runtime, leading to backtracking. In practice, this rule should not be necessary for useful programs, as it indicates the presence of dead code. However, it is absolutely essential in the theory to ensure preservation in the presence of predicate calls.

$$\frac{\Gamma(r_1) = \Im(M_1 : a) \quad \Gamma(r_2) = \Im(M_2 : a) \quad \Delta \vdash M_1 \sqcap M_2 = \sigma \quad [[\sigma]]\Delta; [\sigma]\Gamma \vdash [\sigma]I\ \mathsf{ok}}{\Delta; \Gamma \vdash \mathtt{get\_val}\ r_1, r_2; I\ \mathsf{ok}}\ \textsc{Getval-S} \qquad \frac{\Delta \vdash M_1 \sqcap M_2 = \bot \quad \Gamma(r_1) = \Im(M_1 : a) \quad \Gamma(r_2) = \Im(M_2 : a)}{\Delta; \Gamma \vdash \mathtt{get\_val}\ r_1, r_2; I\ \mathsf{ok}}\ \textsc{Getval-F}$$

All unification in T-Prolog and TWAM is first-order, thus the unification judgements $\Delta \vdash M_1 \sqcap M_2 = \sigma$ and $\Delta \vdash M_1 \sqcap M_2 = \bot$ correspond closely to standard algorithms in the literature [23]. As in dynamic unification, unification uses auxilliary occurs-check judgements $x \in M$ and $x \notin M$. Substitutions in TWAM are capture-avoiding and simultaneous. For example, we write $[M_1/x_1, M_2/x_2]$ for a simulateous subtitution on x_1 and x_2 or $[\sigma_1, \sigma_2]$ for simultaneous composition of arbitrary substitutions σ_1, σ_2:

$$\frac{}{\Delta \vdash x \sqcap x = \cdot}\ \sqcap\cdot \qquad \frac{x \notin M}{\Delta \vdash x \sqcap M = [M/x]}\ \sqcap x1 \qquad \frac{x \notin M}{\Delta \vdash M \sqcap x = [M/x]}\ \sqcap x2 \qquad \frac{x \neq x' \quad x' \notin x}{x \notin x'}\ \notin x \qquad \frac{x \notin M_i(\forall i)}{x \notin c\ \vec{M}}\ \notin x\ \vec{M}$$

$$\frac{\begin{array}{c}\Delta \vdash M_1 \sqcap M_1' = \sigma_1 \\ \vdots \\ [\sigma_{n-1}, \ldots, \sigma_1]\Delta \vdash [\sigma_{n-1}, \ldots, \sigma_1]M_n \sqcap [\sigma_{n-1}, \ldots, \sigma_1]M_n' = \sigma_n\end{array}}{\Delta \vdash c\ M_1\ \ldots\ M_n \sqcap c\ M_1'\ \ldots\ M_n' = \sigma_n, \ldots, \sigma_1}\ \sqcap c \qquad \frac{}{x \in x}\ \in x$$

$$\frac{\begin{array}{c}\Delta \vdash M_1 \sqcap M_1' = \sigma_1 \\ \vdots \\ [\sigma_{i-1}, \ldots, \sigma_1]\Delta \vdash [\sigma_{i-1}, \ldots, \sigma_1]M_i \sqcap [\sigma_{i-1}, \ldots, \sigma_1]M_i' = \bot\end{array}}{\Delta \vdash c\ M_1\ \ldots\ M_n \sqcap c\ M_1'\ \ldots\ M_n' = \bot}\ \bot c1 \qquad \frac{x \in M_i(\exists i)}{x \in c\ \vec{M}}\ \in x\ \vec{M}$$

$$\frac{c \neq c'}{\Delta \vdash c\ M_1\ \ldots\ M_n \sqcap c'\ M_1'\ \ldots\ M_m' = \bot}\ \bot c2 \qquad \frac{x \in M}{\Delta \vdash x \sqcap M = \bot}\ \bot x1 \qquad \frac{x \in M}{\Delta \vdash M \sqcap x = \bot}\ \bot x2$$

6.1.5 Spines. Recall that a Prolog spine serves to unify some terms $M_1 \sqcap M_2$, the distinction being that unlike in `get_val`, the outermost shape of M_2 is known statically. As above, this unification must be made explicit in the type system. As before, a spine type expresses a typing precondition on each unificand and a typing postcondition. Previously the postcondition was trivial, but in our generalized *dependent spine types*, the postcondition says that some unification problem $\Delta \vdash M_1 \sqcap M_2$ has succeeded. We write dependent spine types as $\Pi x_1 : a_1. \cdots \Pi x_n : a_n.\ (M_1 \sqcap M_2)$ to say that M_1 and M_2 will be unified if the spine succeeds, where the x_i stand for the unificand subterms associated with each instruction of the spine.

In Putstr, we temporarily introduce a fresh LF variable x for our new Prolog term, which is then unified with the concrete term resulting from the spine. In Getstr, the unificand is the existing term stored in r. In Unifyvar we extend Δ with a fresh unification variable standing for the given argument (because this variable may be needed later in a proof term), while in Unifyval we do not extend Δ but rather supply an existing term as the spinal argument. At the end of the spine, if the terms unify, then the rule $\sqcap\sigma$ applies the unifier σ while typechecking I, else $\sqcap\bot$ says

typechecking is vacuous because unification will fail at runtime. As in `get_val`, if $\sqcap\bot$ applies at compile-time, it indicates the presence of dead code, but it is of essential use in the preservation proof.

$$\frac{\Sigma(c) = \vec{a} \to a \quad \Delta, x : a; \Gamma\{r : \mathfrak{S}(x : a)\} \vdash I :_s \Pi\vec{x} : \vec{a}.\ (x \sqcap c\ \vec{x})}{\Delta; \Gamma \vdash \texttt{put\_str}\ c, r; I\ \texttt{ok}}\ \textsc{Putstr} \qquad \frac{\Sigma(c) = \vec{a} \to a \quad \Gamma(r) = \mathfrak{S}(M : a) \quad \Delta; \Gamma \vdash I :_s \Pi\vec{x} : \vec{a}.\ (M \sqcap c\ \vec{x})}{\Delta; \Gamma \vdash \texttt{get\_str}\ c, r; I\ \texttt{ok}}\ \textsc{Getstr}$$

$$\frac{\Delta, x : a; \Gamma\{r : \mathfrak{S}(x : a)\} \vdash I :_s J}{\Delta; \Gamma \vdash \texttt{unify\_var}\ r, x : a.\ I :_s \Pi x : a.\ J}\ \textsc{Unifyvar} \qquad \frac{\Gamma(r) = \mathfrak{S}(M : a) \quad \Delta; \Gamma \vdash [M/x]I :_s [M/x]J}{\Delta; \Gamma \vdash \texttt{unify\_val}\ r, x : a.\ I :_s \Pi x : a.\ J}\ \textsc{Unifyval}$$

$$\frac{\Delta \vdash M_1 \sqcap M_2 = \sigma \quad [\![\sigma]\!]\Delta; [\sigma]\Gamma \vdash [\sigma]I\ \texttt{ok}}{\Delta; \Gamma \vdash I :_s M_1 \sqcap M_2}\ \sqcap\sigma \qquad \frac{\Delta \vdash M_1 \sqcap M_2 = \bot}{\Delta; \Gamma \vdash I :_s (M_1 \sqcap M_2)}\ \sqcap\bot$$

6.1.6 Environments. The typing rules for environment tuples are unchanged, since tuples are orthogonal to Prolog terms and LF in general:

$$\frac{\Delta; \Gamma \vdash I :_t (\vec{\tau} \to \{r_d : \mathrm{x}[\vec{\tau}]\}) \quad (\text{where } n = |\vec{\tau}|)}{\Delta; \Gamma \vdash \texttt{put\_tuple}\ r_d, n; I\ \texttt{ok}}\ \textsc{PutTuple}$$

$$\frac{\Gamma(r_s) = \mathrm{x}[\vec{\tau}] \quad \Gamma\{r_d : \tau_i\} \vdash I\ \texttt{ok} \quad (\text{where } i \le |\vec{\tau}|)}{\Delta; \Gamma \vdash \texttt{proj}\ r_d, r_s, i; I\ \texttt{ok}}\ \textsc{Proj} \qquad \frac{\Gamma(r) = \tau \quad \Delta; \Gamma \vdash I :_t J}{\Delta; \Gamma \vdash \texttt{set\_val}\ r; I :_t (\tau \to J)}\ \textsc{SetVal}$$

6.2 Code and Heap Value Typing Invariants

In dependent TWAM, code values can accept LF terms as arguments, as reflected in Code. Furthermore, because heap values can now have dependent types, the heap value typing judgement is now $\Delta; \Gamma \vdash v^H : \tau$, where the added context Δ contains an LF variable for each free variable on the heap. The rule Close is generalized to close over LF terms, while the rules **FREE**, **BOUND** , and $c\langle\rangle$ are generalized to singleton types. As with register files, tuples and closures enforce that words are canonical for simplicity:

$$\frac{\Delta; \Psi \vdash w_{env} : \tau \quad \Delta; \Psi \vdash \ell^C\ \vec{M} : \Pi\vec{x} : \vec{A}.\ \neg\Gamma\{\mathsf{env} : \tau\} \quad w_{env}\ \ \mathsf{canon}}{\Delta; \Psi \vdash \mathsf{close}\ (w_{env}, \ell^C\ \vec{M}) : \Pi\vec{x} : \vec{A}.\ \neg\Gamma}\ \textsc{Close}$$

$$\frac{\Delta; \Psi \vdash w_1 : \tau_1 \quad w_1\ \ \mathsf{canon} \quad \cdots \quad \Delta; \Psi \vdash w_n : \tau_n \quad w_n\ \ \mathsf{canon}}{\Delta; \Psi \vdash \langle w_1, \ldots, w_n\rangle : \mathrm{x}[\tau_1, \ldots, \tau_n]}\ \langle\rangle$$

$$\frac{\Delta(x) = a}{\Delta; \Psi \vdash \mathbf{FREE}[x : a] : \mathfrak{S}(x : a)}\ \mathbf{FREE} \qquad \frac{\Delta; \Psi \vdash \ell^H : \mathfrak{S}(M : a)}{\Delta; \Psi \vdash \mathbf{BOUND}\ \ell^H : \mathfrak{S}(M : a)}\ \mathbf{BOUND}$$

$$\frac{\Sigma(c) = \vec{a} \to a \quad \Delta; \Psi \vdash \ell_i^H : \mathfrak{S}(M_i : a_i)}{\Delta; \Psi \vdash_{\Sigma;\Xi} c\langle\ell_1^H, \ldots, \ell_n^H\rangle : \mathfrak{S}(c\ \vec{M} : a)}\ c\langle\rangle \qquad \frac{(\vec{x} : \vec{A}); \Gamma \vdash I\ \texttt{ok}}{\cdot \vdash \mathsf{code}[\vec{x} : \vec{A}.\ \Gamma](\lambda\vec{x} : \vec{A}.\ I) : \Pi\vec{x} : \vec{A}.\ \neg\Gamma}\ \textsc{Code}$$

6.3 Machine Typing Invariants

The runtime behavior of a TWAM program does not depend on type information, i.e. TWAM is easily executed by first type-erasing it to SWAM and then executing the SWAM program. However, just as TWAM adds typing and proof term annotations to instructions, our theoretical presentation of the machine states is annotated with LF variables and proof terms, as well.

LF Contexts and Mappings. When we prove soundness for TWAM (Theorem 1) in Section 6.4, we will show that for each successful execution trace, an LF proof term exists *in some context* Δ. For convenience, we make that context an additional field of the machine state, but this is not strictly necessary because it contains one variable for each free variable in the heap H and could

thus be computed as a function of H. For Theorem 1 to be meaningful, it is essential that Δ only contains Prolog terms and not arbitrary LF propositions. Otherwise, if we wished to find a proof term for some query A, we could simply add A to the context with `put_var` and obtain a trivial "proof". Consider the following example (which assumes we have successfully defined the Riemann Hypothesis in Prolog):

```
put_var r1, x:Riemann_hypothesis.
succeed[x:Riemann_hypothesis]
```

Luckily, we easily enforce that Δ contains only Prolog terms by adding a syntactic restriction in `put_var`.

The addition of LF variables affects the heap as well: free variables are now annotated as $\mathbf{FREE}[x : a]$ because they are in correspondence with LF variables x. As a technical device to support our progress and preservation theorems, we maintain the invariant that this correspondence is unique with a mapping μ between each variable and its unique location on the heap.

$$\text{LF Mappings } \mu ::= \cdot \mid x@(\ell^H : a), \mu$$

The syntax $x@(\ell^H : a)$ says the LF variable x has type a and is located at ℓ^H. The judgement $\cdot \vdash (\Delta; \mu) : H$ says that μ correctly mediates Δ and H (i.e. assigns a unique location in H to each variable of Δ):

$$\frac{\Delta = \cdot \quad \mu = \cdot}{\cdot \vdash (\Delta; \mu) : \{\}}\ \mu\text{-Nil} \qquad \frac{\cdot \vdash (\Delta; \mu) : H \quad v \neq \mathbf{FREE}[x : a]}{\cdot \vdash (\Delta; \mu) : H\{\{\ell^H \mapsto v\}\}}\ \mu\text{-Skip}$$

$$\frac{\cdot \vdash (\Delta; \mu) : H}{\cdot \vdash (\Delta, x : a; \mu :: (x@\ell^H : a)) : H\{\{\ell^H \mapsto \mathbf{FREE}[x : a]\}\}}\ \mu\text{-Cons}$$

Trails. Trails are generalized in two straightforward ways. First, failure continuations are now allowed to close over LF terms. Second, trail typing annotations $\ell^H : a$ are now generalized to remember the corresponding LF variable name ($x@\ell^H : a$) so that Δ can be updated accordingly in unwinding:

$$\frac{\begin{array}{c}\mathtt{unwind}(S, \Delta, t) = (\Delta'; S') \quad \Delta'; S' \vdash T'\ \mathtt{ok} \quad \Delta'; \Psi' \vdash w : \tau \\ \cdot \vdash (\Delta'; \mu') : H' \quad \Delta' \vdash S' : (\Xi, \Psi') \quad \Delta'; \Psi' \vdash \ell^C\ \vec{M} : \neg\{\mathtt{env} : \tau\}\end{array}}{\Delta; S \vdash (t, w, \ell^C\ \vec{M}) :: T'\ \mathtt{ok}}\ \textsc{Trail-Cons} \qquad \frac{}{\Delta; S \vdash \epsilon\ \mathtt{ok}}\ \textsc{Trail-Nil}$$

$$\mathtt{unwind}(S, \Delta, (x@\ell^H : a) :: t) = \mathtt{unwind}(S\{\ell^H \mapsto \mathbf{FREE}[x : a]\}, (\Delta, x : a), t) \qquad \mathtt{unwind}(S, \Delta, \epsilon) = (\Delta, S)$$
$$\mathtt{update\_trail}(x@\ell^H : a, (t, w_{env}, \ell^C) :: T) = ((x@\ell^H : a) :: t, w_{env}, \ell^C) :: T \qquad \mathtt{update\_trail}((x@\ell^H : a), \epsilon) = \epsilon$$

Register File Types. Register file typing now requires that words are canonical, for the sake of simplicity:

$$\frac{\Delta; \Psi \vdash w_1 : \tau_1 \quad w_1\ \mathtt{canon} \quad \cdots \quad \Delta; \Psi \vdash w_n : \tau_n \quad w_n\ \mathtt{canon}}{\Delta; \Psi \vdash \{r_1 \mapsto w_1, \ldots, r_n \mapsto w_n\} : \{r_1 : \tau_1, \ldots, r_n : \tau_n\}}\ \text{RF}$$

Machine States. The machine state typing invariants are updated to use the dependent forms of existing judgements in addition to the new invariant $\cdot \vdash (\Delta, \mu) : H$. As before, spinal states each appeal to an auxilliary invariant.

$$\frac{\Delta;(C,H) \vdash T\ \mathsf{ok} \quad \cdot \vdash (\Delta,\mu) : H \quad \Delta \vdash (C,H) : (\Xi;\Psi) \quad \Delta;\Psi \vdash R : \Gamma \quad \Delta;\Gamma \vdash I\ \mathsf{ok}}{\cdot \vdash (T,\Delta,(C,H),R,I)\ \mathsf{ok}}\ \textsc{Mach}$$

$$\frac{\Delta;(C,H) \vdash T\ \mathsf{ok} \quad \cdot \vdash (\Delta,\mu) : H \quad \Delta;\Psi \vdash R : \Gamma \quad \Delta \vdash (C,H) : (\Xi;\Psi) \quad \Delta;\Gamma \vdash I :_t J \quad \Delta;\Psi \vdash (\vec{w},r,n)\ \mathtt{writes}\ J}{\cdot \vdash \mathsf{twrite}(T,\Delta,(C,H),R,I,\vec{w},r,n)\ \mathsf{ok}}\ \textsc{Mach-TWrite}$$

$$\frac{\Delta;(C,H) \vdash T\ \mathsf{ok} \quad \cdot \vdash (\Delta,\mu) : H \quad \Delta;\Psi \vdash R : \Gamma \quad \Delta \vdash (C,H) : (\Xi;\Psi) \quad \Delta;\Gamma \vdash I :_s J \quad \Delta;\Psi \vdash \vec{\ell}^H\ \mathtt{reads}\ J}{\cdot \vdash \mathsf{read}(T,\Delta,(C,H),R,I,\vec{\ell}^H)\ \mathsf{ok}}\ \textsc{Mach-Read}$$

$$\frac{\Delta;(C,H) \vdash T\ \mathsf{ok} \quad \cdot \vdash (\Delta,\mu) : H \quad \Delta;\Psi \vdash R : \Gamma \quad \Delta \vdash (C,H) : (\Xi;\Psi) \quad \Delta;\Gamma \vdash I :_s J \quad \Delta;\Psi \vdash (\vec{\ell}^H,\ell^H,c)\ \mathtt{writes}\ J}{\cdot \vdash \mathsf{write}(T,\Delta,(C,H),R,I,c,\ell^H,\vec{\ell}^H)\ \mathsf{ok}}\ \textsc{Mach-Write}$$

However, the auxilliary invariants for Prolog spines have become more complex. The read spine invariant considers a term sequence $\vec{M}$ for the arguments already read and a second sequence $\vec{M'}$ for those remaining. The invariant holds if (a) every x_i can still unify with M'_i and (b) every ℓ^H_i has the type expected by the spine type. The write spine invariant requires that (a) the destination ℓ^H matches the result type of the constructor c, (b) the existing arguments $\vec{\ell^H}$ match the initial argument types, and (c) the remainder of the spine matches the remaining argument types.

$$\frac{\Delta;\Psi \vdash \vec{\ell}^H : \vec{\tau}_1 \quad |\vec{\tau}_2| = n}{\Delta;\Psi \vdash (n,\, r,\, \vec{\ell}^H)\ \mathtt{writes}\ (\vec{\tau}_2 \to \{r : \mathrm{x}[\vec{\tau}_1\vec{\tau}_2]\})}\ \textsc{TWrites}$$

$$\frac{\Delta \vdash c\ \vec{M}\ \vec{x} \sqcap c\ \vec{M}\ \vec{M'} = \sigma \quad \Delta;\Psi \vdash \ell^H_i : \Im(M'_i : [M'_1,\ldots,M'_{i-1}/x_1,\ldots,x_{i-1}]A_i)}{\Delta;\Psi \vdash \vec{\ell}^H\ \mathtt{reads}\ \Pi\vec{x} : \vec{A}.(c\ \vec{M}\ \vec{M'} \sqcap c\ \vec{M}\ \vec{x})}\ \textsc{Reads}$$

$$\frac{\Sigma(c) = \vec{a}_1 \to \vec{a}_2 \to a \quad \Delta;\Psi \vdash \vec{\ell}^H : \Im(\vec{M} : \vec{a}_1) \quad \Psi(\ell^H) = \Im(x' : a)}{\Delta;\Psi \vdash (\vec{\ell}^H,\, \ell^H,\, c)\ \mathtt{writes}\ \Pi\vec{x} : \vec{a}_2.x' \sqcap c\ \vec{M}\ \vec{x}}\ \textsc{Writes}$$

6.4 Metatheory

In the dependent setting, we show our primary result that all TWAM programs are sound proof search procedures in the following sense:

Theorem 1 (Soundness). *If* $\cdot \vdash m\ \mathsf{ok}$ *and* $m \longmapsto^* m'$ *and* $m'\ \mathsf{done}$ *then* $m' = (T,\Delta,S,R,\mathsf{succeed}[M : A];I)$ *and* $\Delta \vdash M : A$.

This theorem is an immediate corollary of progress and preservation, by inversion on the typing rule for succeed.

Thus it suffices to show progress and preservation and their supporting lemmas. We present here only the lemmas that are new or significantly different from the simply-typed versions. A detailed proof for the dependently-typed system is in the electronic appendix.

6.4.1 *Static Occurs Check.*

Lemma 14 (Static Occurs Check Totality). *For all terms* M *and all variables* x, *either* $x \in M$ *or* $x \notin M$

Proof. By induction on the structure of M. □

6.4.2 Static Unification.

Lemma 15 (Static Unification Totality). *For all LF terms M_1, M_2, if $\Delta \vdash M_2 : A$ and $\Delta \vdash M_1 : A$ then $\Delta \vdash M_1 \sqcap M_2 = \sigma$ or $\Delta \vdash M_1 \sqcap M_2 = \bot$.*

Proof. By lexicographic induction on $|\Delta|$ and the structure of M_1. The base cases hold by Lemma 14. The inductive case $M_1 = c\ \vec{M}$, $M_2 = c\ \vec{M'}$ (where $|M| = |M'|$) relies on a subclaim:

claim 2. *For each $1 \leq i \leq |\vec{M}|$, consider $\sigma = [\sigma_{i-1}, \ldots, \sigma_1]$. Then $[\sigma]\Delta \vdash [\sigma]M_i \sqcap M'_i = \sigma_i$ for some σ_i or $[\sigma]\Delta \vdash [\sigma]M_i \sqcap [\sigma]M'_i = \bot$.*

Proof. By cases on i.
case $i = 1$

By IH because M_1 is structurally smaller than $c\ \vec{M}$.
case $i > 1$

By IH: if $\sigma = \cdot$ then M_i is structurally smaller than $c\ \vec{M}$, else $\sigma = [\vec{M''}/\vec{x}]$ where $x_i \in Dom(\Delta)$ and thus $|[\sigma]\Delta| < \Delta$. □

□

Lemma 16 (Static Unification Correctness). *If $\Delta \vdash M : A$ and $\Delta \vdash M' : A$ and $\Delta \vdash M \sqcap M' = \sigma$, then*

- $[\sigma]M = [\sigma]M'$
- *For all substitutions σ', if $[\sigma']M = [\sigma']M'$ then there exists some σ^* such that $\sigma' \equiv_\alpha \sigma^*, \sigma$.*

Proof. Analogous to standard results from the literature. □

As a technical device for Lemma 22 we introduce a judgement $M_1 \sqsubset M_2$ meaning "M_1 is a strict substructure of M_2":

$$\frac{}{M_i \sqsubset c\ \vec{M}}\ \sqsubset\text{-Base} \qquad \frac{M \sqsubset M_i}{M \sqsubset c\ \vec{M}}\ \sqsubset\text{-Ind}$$

The following lemmas support the proof of Lemma 22.

Lemma 17 (Occurs to Substructure). *If $x \in M$ and $x \neq M$ then $x \sqsubset M$*

Proof. By induction on the derivation $x \in M$. □

Lemma 18 (Substructure to Occurs). *If $x \sqsubset M$ then $x \in M$.*

Proof. By induction on the derivation $x \sqsubset M$. □

Lemma 19 (Substitution Preserves Substructure). *If $M_1 \sqsubset M_2$ then $[M/x]M_1 \sqsubset [M/x]M_2$*

Proof. By induction on the derivation $M_1 \sqsubset M_2$. □

Lemma 20 (Transitivity of Substructure). *If $M_1 \sqsubset M_2$ and $M_2 \sqsubset M_3$ then $M_1 \sqsubset M_3$.*

Proof. By induction on the derivation $M_2 \sqsubset M_3$. □

Lemma 21 (Substructures Don't Unify). *If $\Delta \vdash M : A, \Delta \vdash M' : A$ and $M \sqsubset M'$ then $\Delta \vdash M \sqcap M' = \bot$.*

Proof. By lexicographic induction on $|\Delta|$ and the structure of M.

Consider the cases for $M \sqsubset M'$. The case $x \sqsubset c\ \vec{M}$ holds by Lemma 18.

case $\dfrac{(\text{because } c\ \vec{M} = M_i')}{c\ \vec{M} \sqsubset c'\ \vec{M'}}\ \sqsubset\text{-Base}$

If $c' \neq c$, then unification fails immediately, so assume $c \neq c'$. Observe $M_i \sqsubset c\ \vec{M}$ by rule $\sqsubset$-Base. Since $M_i' = c\ \vec{M}$, we have $M_i \sqsubset M_i'$. By Lemma 19, for any substitution σ, we have $[\sigma]M \sqsubset [\sigma]M'$. Note that when we unify $c\ \vec{M}$ and $c\ \vec{M'}$ we either fail before M_i or attempt to compute $[\sigma]M \sqcap [\sigma]M'$ for some σ. If we failed already, the case is done. If we succeeded, then by the IH $[\sigma]\Delta \vdash [\sigma]M \sqcap [\sigma]M' = \bot$ and we fail here.

case $\dfrac{c\ \vec{M} \sqsubset M_i'}{c\ \vec{M} \sqsubset c'\ \vec{M'}}\ \sqsubset\text{-Ind}$

As in the previous case, AWLOG $c = c'$. Now since $c\ \vec{M} \sqsubset c\ \vec{M'}$ then by Lemma 20, $M_i \sqsubset M_i'$ for the i such that $c\ \vec{M} \sqsubset M_i$. The rest of the case is analogous to the last one. □

Lemma 22 (Unification Lemma of Doom). *Unifications that fail are doomed to fail forever. That is, if* $\Delta, x : A \vdash M_1 \sqcap M_2 = \bot$ *and* $\Delta \vdash M : A$ *then* $[M/x]\Delta \vdash [M/x]M_1 \sqcap [M/x]M_2 = \bot$.

Proof. By lexicographic induction on $|\Delta|$ and the unification derivation $\Delta, x : A \vdash M_1 \sqcap M_2 = \bot$.

case $\dfrac{x' \neq M_2 \quad x' \in M_2}{\Delta, x : A \vdash x' \sqcap M_2 = \bot}\ \bot x1$

Case on whether $x = x'$.

case $x \neq x'$

In this case, $[M/x]x' = x'$ and $x' \in [M/x]M_2$ so the unification still fails.

case $x = x'$

This case reduces to the following claim:

Claim 3. *If* $x \in M$ *and* $\Delta \vdash M' : A$ *and* $\Delta, x : A \vdash M : A$ *then* $[M/x]\Delta \vdash M' \sqcap [M/x] = \bot$.

Proof. By Lemma 17, $x \sqsubset M$. By Lemma 19, $M' \sqsubset [M'/x]M$. By Lemma 21, $[M'/x]\Delta \vdash M' \sqcap [M'/x]M = \bot$. □

case $\dfrac{x' \neq M \quad x' \in M}{\Delta, x : A \vdash M \sqcap x' = \bot}\ \bot x2$

This case holds by symmetry.

case $\dfrac{c \neq c'}{\Delta, x : A \vdash c\ M_1\ \ldots\ M_n \sqcap c'\ M_1'\ \ldots\ M_m' = \bot}\ \bot c2$

This case holds because substitution preserves head constructors.

case $\dfrac{\begin{array}{c} M_1 \sqcap M_1' = \sigma_1 \\ \vdots \\ [\sigma_{i-1}, \ldots, \sigma_1]\Delta \vdash [\sigma_{i-1}, \ldots, \sigma_1]M_i \sqcap [\sigma_{i-1}, \ldots, \sigma_1]M_i' = \bot \end{array}}{\Delta \vdash c\ M_1\ \ldots\ M_n \sqcap c\ M_1'\ \ldots\ M_n' = \bot}\ \bot c1$

If some $i' < i$ fails to unify, we're done. Otherwise we attempt to unify $[\sigma]M_i \sqcap [\sigma]M_2$ where $\sigma = \sigma'_{i-1}, \ldots, \sigma'_1, M/x$. By Lemma 16, $\sigma = \sigma^*, \sigma_{i-1}, \ldots, \sigma_1$ so we can apply the IH to get and

$$\dfrac{\begin{array}{c}[M/x]M_1 \sqcap [M/x]M'_1 = \sigma'_1 \\ \vdots \\ [\sigma'_{i-1}, \ldots, \sigma'_1, M/x]\Delta \vdash [\sigma'_{i-1}, \ldots, \sigma'_1, M/x]M_i \sqcap [\sigma'_{i-1}, \ldots, \sigma'_1, M/x]M'_i = \bot\end{array}}{[M/x]\Delta \vdash [M/x]c\ M_1\ \ldots\ M_n \sqcap [M/x]c\ M'_1\ \ldots\ M'_n = \bot}\ \bot c1$$

□

6.4.3 *Substitution.*

Lemma 23 (Substitution). *All appropriate typing judgements support substitution.*

(1) LF terms: *If* $\Delta, x\colon A \vdash M_1 \colon A'$, $\Delta \vdash M_2 \colon A$ *then* $[M_2/x]\Delta \vdash [M_2/x]M_1 \colon [M_2/x]A$.
(2) Operands: *If* $\Delta_1, x\colon A; \Delta_2; \Gamma \vdash op\colon \tau, \Delta_1 \vdash M\colon A$ *then* $\Delta_1, [M/x]\Delta_2; [M/x]\Gamma \vdash [M/x]op\colon [M/x]\tau$.
(3) Word values: *If* $\Delta_1, x\colon A, \Delta_2; \Psi \vdash w\colon \tau, \Delta_1 \vdash M\colon A$ *then* $\Delta_1, [M/x]\Delta_2; [M/x]\Psi \vdash [M/x]w\colon [M/x]\tau$.
(4) Register Files: *If* $\Delta_1, x\colon A, \Delta_2; \Psi \vdash R\colon \Gamma, \Delta_1 \vdash M : A$ *then* $\Delta_1, [M/x]\Delta_2; [M/x]\Psi \vdash [M/x]R\colon [M/x]\Gamma$.
(5) Heap values: *If* $\Delta_1, x\colon a, \Delta_2; \Psi \vdash v^H\colon \tau, \Delta_1 \vdash M\colon a$, $v^H \neq \textbf{\textit{FREE}}[x\colon a]$ *then* $\Delta_1, [M/x]\Delta_2; [M/x]\Psi \vdash [M/x]v^H\colon [M/x]\tau$
(6) Basic blocks: *If* $\Delta_1, x\colon A, \Delta_2; \Gamma \vdash I$ ok, $\Delta_1 \vdash M\colon A$ *then* $\Delta_1, [M/x]\Delta_2; [M/x]\Gamma \vdash [M/x]I$ ok.

Proof. Each claim holds by lexicographic induction on $|\Delta|$ and the structure of the typing derivation. The first five claims are straightforward. The interesting cases of the final claim are the unification instructions, because there is a subtle interaction between static and dynamic unification. The case for `get_val` r_1, r_2 is representative:

case $\dfrac{[\sigma]\Delta_1, x : A, \Delta_2; [\sigma]\Gamma \vdash [\sigma]I\ \mathsf{ok} \quad \Delta_1, x : A, \Delta_2 \vdash M_1 \sqcap M_2 = \sigma \quad \Gamma(r_1) = \Im(M_1 : a) \quad \Gamma(r_2) = \Im(M_2 : a)}{\Delta_1, x : A, \Delta_2; \Gamma \vdash \texttt{get\_val}\ r_1, r_2; I\ \mathsf{ok}}$ GetVal-S

By claim 1, $\Delta_1, [M/x]\Delta_2 \vdash [M/x]M_1 : a$ and $\Delta_1, [M/x]\Delta_2 \vdash [M/x]M_2 : a$ so by Lemma 15, either $\Delta_1, [M/x]\Delta_2 \vdash [M/x]M_1 \sqcap [M/x]M_2 = \sigma'$ or $\Delta_1, [M/x]\Delta_2 \vdash [M/x]M_1 \sqcap [M/x]M_2 = \bot$.
subcase $\Delta_1, [M/x]\Delta_2 \vdash [M/x]M_1 \sqcap [M/x]M_2 = \sigma'$

By Lemma 16, $[\sigma']([M/x]M_1) = [\sigma']([M/x]M_2)$ which we can rewrite as $[\sigma', M/x]M_1 = [\sigma', M/x]M_2$. Also by Lemma 16, σ is a most general unifier of M_1 and M_2. Thus there exists σ^* such that $\sigma', M/x = \sigma^*, \sigma$ (they need not be syntactically equal, but must be alpha-equivalent). In particular, alpha-vary σ^* such that it substitutes for x. Then by iterating the IH (we can do this because $|\Delta|$ decreases every time), $[\sigma^*, \sigma](\Delta_1, x : A, \Delta_2); [\sigma^*, \sigma]\Gamma \vdash [\sigma^*, \sigma]I$ ok. By the assumption that σ^*, σ substitutes for x, we have $[\sigma^*, \sigma](\Delta_1, \Delta_2); [\sigma^*, \sigma]\Gamma \vdash [\sigma^*, \sigma]I$ ok which suffices to show the result:

$$\dfrac{\begin{array}{cc}([M/x]\Gamma)(r_1) = \Im([M/x]M_1 : a) & ([M/x]\Gamma)(r_2) = \Im([M/x]M_2 : a) \\ \Delta_1, [M/x]\Delta_2 \vdash [M/x]M_1 \sqcap [M/x]M_2 = \sigma^*, \sigma & [\sigma^*, \sigma]\Delta; [\sigma^*, \sigma]\Gamma \vdash [\sigma^*, \sigma]I\ \mathsf{ok}\end{array}}{\Delta_1, [M/x]\Delta_2; [M/x]\Gamma \vdash \texttt{get\_val}\ r_1, r_2; [M/x]I\ \mathsf{ok}}$$ GetVal-S

subcase $\Delta_1, [M/x]\Delta_2 \vdash [M/x]M_1 \sqcap [M/x]M_2 = \bot$

In this case, since the unification failed, the result is vacuously well-typed:

$$\dfrac{([M/x]\Gamma)(r_1) = \Im([M/x]M_1 : a) \quad ([M/x]\Gamma)(r_2) = \Im([M/x]M_2 : a) \quad \Delta_1, [M/x]\Delta_2 \vdash [M/x]M_1 \sqcap [M/x]M_2 = \bot}{\Delta_1, [M/x]\Delta_2; [M/x]\Gamma \vdash \texttt{get\_val}\ r_1, r_2; [M/x]I\ \mathsf{ok}}$$ GetVal-F

$$\textbf{case}\ \dfrac{\Gamma(r_1) = \Im(M_1 : a) \quad \Gamma(r_2) = \Im(M_2 : a) \quad \Delta_1, x : A, \Delta_2 \vdash M_1 \sqcap M_2 = \bot}{\Delta_1, x : A, \Delta_2; \Gamma \vdash \texttt{get\_val}\ r_1, r_2; I\ \mathsf{ok}}$$

By Lemma 22. □

6.4.4 *Soundness of Unification.*

Lemma 24 (Soundness of unify). *If* $\Delta \vdash M_1 : a$, $\Delta \vdash M_2 : a$, $\Delta \vdash S : (\Xi; \Psi)$, $\Delta; S \vdash T$ ok, $\cdot \vdash (\Delta; \mu) : S$, $\Delta; \Psi \vdash \ell_1^H : \mathfrak{S}(M_1 : A)$, $\Delta; \Psi \vdash \ell_2^H : \mathfrak{S}(M_2 : A)$ *then*

- *If* $\Delta \vdash M_1 \sqcap M_2 = \bot$ *then* $\mathtt{unify}(\Delta, S, T, \ell_1^H, \ell_2^H) = \bot$
- *If* $\Delta \vdash M_1 \sqcap M_2 = \sigma$ *then* $\mathtt{unify}(\Delta, S, T, \ell_1^H, \ell_2^H) = (\Delta', S', T')$ *where* $\Delta' = [\sigma]\Delta$ *and* $[\sigma]\Delta \vdash H' : [\sigma]\Psi$ *and* $\Delta', S' \vdash T'$ ok.

Proof. As in SWAM, with the additional use of Lemma 23. □

6.4.5 *Words and Operands.*

Lemma 25 (Word Totality). *If* $\Delta; \Psi \vdash w : \tau$ *then* $w \Downarrow w'$ *and* w' canon.

Proof. By induction on $size(w)$, defined by $size(\ell) = 0$, $size(w\ M) = 1 + size(w)$, $size(\lambda x : A.w) = 1 + size(w)$, appealing to Lemma 23 and the fact that substitution preserves $size(M)$. □

Lemma 26 (Operand Resolution). *For all operands* op, *if* $\Delta; \Gamma \vdash_\Xi op : \tau$ *and* $\Delta; \Psi \vdash R : \Gamma$ *then* $R \vdash op \rightsquigarrow w$ *for some word* w *and* $\Delta; \Psi \vdash_\Xi w : \tau$.

Proof. By induction on $\Delta; \Gamma \vdash_\Xi op : \tau$ and inversion on register file typing. □

Lemma 27 (Word Preservation). *If* $\Delta; \Psi \vdash w : \tau$ *and* $w \Downarrow w'$ *then* $\Delta; \Psi \vdash w' : \tau$.

Proof. By induction on the trace $w \Downarrow w'$ and Lemma 23. □

Lemma 28 (Operand Preservation). *If* $\Delta; \Psi \vdash R : \Gamma$ *and* $\Delta; \Gamma \vdash op : \tau$ *and* $R \vdash op \Downarrow w$ *then* w canon *and* $\Delta; \Psi \vdash w : \tau$.

Proof. Follows directly from Lemmas 26 and 27. □

Lemma 29 (Operand Canonicalization). *If* $\Delta; \Psi \vdash R : \Gamma$ *and* $\Delta; \Gamma \vdash op : \tau$ *then* $R \vdash op \Downarrow w$ *and* w canon *and* $\Delta; \Psi \vdash w : \tau$.

Proof. By Lemma 26, $R \vdash op \Downarrow w'$ and $\Delta; \Psi \vdash w' : \tau$. By Lemma 25, $w' \Downarrow w$, so by rule $op \Downarrow$ we have $R \vdash op \Downarrow w$ and by Lemma 27, $\Delta; \Psi \vdash w : \tau$. □

Lemma 30 (Word Inversion). *If* $\Delta; \Psi \vdash \ell\ \vec{M} : \tau$ *then* $\Delta; \Psi \vdash \ell : \Pi\vec{x} : \vec{A}.\tau'$ *where* $[\vec{M}/\vec{x}]\tau' = \tau$ *and* $\Delta \vdash M_i : [M_1, \ldots, M_{i-1}/x_1, \ldots, x_{i-1}]A_i$.

Proof. By induction on the derivation $\Delta; \Psi \vdash \ell\ \vec{M} : \tau$. □

6.4.6 *Progress and Preservation.*

Theorem 2 (Progress). *If* $\cdot \vdash m$ ok *then either* m done *or* m `fails` *or* $m \longmapsto m'$.

Proof. By cases on m (specifically, cases on I), and by Lemmas 1, 11, 12, 13, 15, 28, 29, and 30. □

Theorem 3 (Preservation). *If* $\cdot \vdash m$ ok *and* $m \longmapsto m'$ *then* $\cdot \vdash m'$ ok.

Proof. By cases on $m \longmapsto m'$, using the assumptions of $\cdot \vdash m$ ok. First consider simulatenously all the cases that end in backtracking. Those cases hold by Lemma 11. Thus it suffices to show the cases that do not backtrack.

$$\textbf{case}\ \frac{R \vdash op \mapsto \ell^H\ \vec{M} \quad C(\ell^H) = \mathtt{close}\ (w, \ell^C\ \vec{M'})}{(T, \Delta, S, R, \mathtt{jmp}\ op; I) \longmapsto (T, \Delta, S, R\{r_1 \mapsto w\}, [\vec{M}\vec{M'}/\vec{x}\vec{x'}]I')}\ \textsc{Jmp}\text{-}\ell^H$$

By Lemma 28, $\Delta; \Psi \vdash \ell^H\ \vec{M} : \neg\Gamma'$. By typing assumption, $\ell^C\ \vec{M} : \Pi\vec{x}' : \vec{A}'.\neg\Gamma''\{r_1 \mapsto \tau\}$ so by Lemma 30, $\Delta; \Psi \vdash \ell^C : \Pi\vec{x}\vec{x}' : \vec{A}\vec{A}'.\neg\Gamma'''$ where $[\vec{M}'/\vec{x}']\Gamma''' = \Gamma''$. By Lemma 1, $C(\ell^C) = \mathsf{code}[\vec{x}'\vec{x} : \vec{A}'\vec{A}](\lambda\vec{x}'\vec{x} : \vec{A}'\vec{A}.I')$ and $\vec{x}'\vec{x} : \vec{A}'\vec{A}; \{r_1 : \tau\} \vdash I'$ ok. By Lemmas 3 and 23, $\Delta; [\vec{M}\vec{M}'/\vec{x}\vec{x}']\Gamma'\{r : \tau\} \vdash [\vec{M}/\vec{x}]I'$ ok where $\Delta; \Psi \vdash w : \tau = [\vec{M}'\vec{M}/\vec{x}'\vec{x}]\tau$. By assumption, $\Delta \vdash \Gamma' \leq \Gamma$, so by Lemma 4, $\Delta; \Gamma\{r_1 : \tau\} \vdash [\vec{M}'\vec{M}/\vec{x}'\vec{x}]I'$ ok. By assumption w canon so $\Delta; \Psi \vdash R\{r_1 \mapsto w\} : \Gamma\{r_1 : \tau\}$, then $\cdot \vdash m'$ ok.

case $$\frac{R \vdash op \mapsto \ell^C\ \vec{M} \quad C(\ell^C) = \mathsf{code}[\vec{x} : \vec{A}.\Gamma'](\lambda\vec{x} : \vec{A}.I')}{(T, \Delta, S, R, \mathtt{jmp}\ op; I) \longmapsto (T, \Delta, S, R, [\vec{M}/\vec{x}]I')}\ \textsc{Jmp}\text{-}\ell^C$$

By Lemma 28, $\Delta; \Psi \vdash \ell^H\ \vec{M} : \neg\Gamma'$. By assumption, $(\vec{x} : \vec{a}); \Gamma' \vdash I'$ ok and by Lemmas 3 and 23, $\Delta; [\vec{M}/\vec{x}]\Gamma' \vdash [\vec{M}/\vec{x}]I'$ ok. Since $\Delta \vdash [\vec{M}/\vec{x}]\Gamma' \leq \Gamma$, by Lemma 4, $\Delta; \Gamma \vdash [\vec{M}/\vec{x}]I'$ ok, so $\cdot \vdash m'$ ok.

case $$\frac{R(r_s) = w}{(T, \Delta, S, R, \mathtt{close}\ r_d, r_s, \ell^C\ \vec{M}; I) \longmapsto (T, \Delta, S\{\{\ell^H \mapsto \mathtt{close}\ (w, \ell^C\ \vec{M})\}\}, R\{r \mapsto \ell^H\}, I)}\ \textsc{Close}$$

By assumption and inversion on $\Gamma(r) = \tau$, we have $\Delta; \Psi \vdash \mathtt{close}\ (w, \ell^C\ \vec{M}) : (\Pi\vec{x} : \vec{A}.\ \neg\Gamma')$, so $\Delta \vdash S\{\{\ell^H \mapsto \mathtt{close}\ [\Gamma'](w, \lambda\vec{x} : \vec{A}.I')\}\} : \Psi\{\{\ell^H : \Pi\vec{x} : \vec{A}.\ \neg\Gamma'\}\}$ and $\Delta; \Psi\{\{\ell^H : \Pi\vec{x} : \vec{A}.\ \neg\Gamma'\}\} \vdash R\{r \mapsto \ell^H\} : \Gamma\{r : \Pi\vec{x} : \vec{A}.\ \neg\Gamma'\}$ and thus, $\cdot \vdash m'$ ok.

case $$\frac{R(r) = w}{(T, \Delta, S, R, \mathtt{push\_bt}\ r, \ell^C\ \vec{M}; I) \longmapsto ((\epsilon, w, \ell^C\ \vec{M}) :: T, \Delta, S, R, I)}\ \textsc{PushBT} \longmapsto$$

By ϵ case of unwind, we have $\mathtt{unwind}(\Delta, S, \epsilon) = (\Delta; S)$ so let $\Delta' = \Delta, S' = S, \mu' = \mu$. By Lemma 1 and inversion, $\Delta'; \Psi' \vdash w : \tau$ and w canon and the rest holds by assumption:

$$\frac{\mathtt{unwind}(\Delta, S, \epsilon) = (\Delta'; S') \quad \Delta'; S' \vdash T\ \mathsf{ok} \quad \cdot \vdash (\Delta'; \mu') : H' \quad \cdot \vdash H' : \Psi' \quad \Delta'; \Psi' \vdash w : \tau \quad \Delta'; \Psi' \vdash \ell^C\ \vec{M} : \neg\{r_1 : \tau\} \quad w\ \mathtt{canon}}{\Delta; S \vdash (\epsilon, w, \ell^C\ \vec{M}) :: T\ \mathsf{ok}}\ \textsc{Trail-Cons}$$

so $\cdot \vdash m'$ ok as well by the assumption $\Delta; \Gamma \vdash I$ ok.

case $$\frac{R(r_1) = w_1 \quad R(r_2) = w_2 \quad \mathtt{unify}(\Delta, S, T, w_1, w_2) = (\Delta', S', T')}{(T, \Delta, S, R, \mathtt{get\_val}\ r_1, r_2; I) \longmapsto (T', \Delta', S', R, I)}\ \textsc{GetVal} \longmapsto$$

By assumption $\Delta \vdash M_1 \sqcap M_2 = \sigma$ where $\Delta; \Psi \vdash w_1 : \Im(M_1 : a)$ and $\Delta; \Psi \vdash w_2 : \Im(M_2 : a)$. By Lemma 13, $\Delta' = [\sigma]\Delta$, $\Delta \vdash S' : [\sigma]\Psi$ and $\cdot \vdash (\Delta', [\sigma]\mu) : H'$ and $\Delta'; S' \vdash T'$ ok. By substitution, $[\sigma]\Delta; [\sigma]\Psi \vdash R : [\sigma]\Gamma$ and by assumption $[\sigma]\Delta; [\sigma]\Gamma \vdash I$ ok so $\cdot \vdash m'$ ok.

case $$\frac{R \vdash op \Downarrow w}{(T, \Delta, S, R, \mathtt{mov}\ r_d, op; I) \longmapsto (T, \Delta, S, R\{r_d \mapsto w\}, I)}\ \textsc{Mov} \longmapsto$$

By Lemma 28, $\Delta; \Psi \vdash w : \tau$ and w canon so $\Delta; \Psi \vdash R\{r_d \mapsto w\} : \Gamma\{r_d : \tau\}$ and $\cdot \vdash m'$ ok.

case $$(T, \Delta, S, R, \mathtt{put\_var}\ \ r, x : a.I) \longmapsto (T, (\Delta, x : a), S\{\{\ell^H \mapsto \mathbf{FREE}[x : a]\}\}, R\{r \mapsto \ell^H\}, I)\ \textsc{PutVar} \longmapsto$$

Let $\mu' = (x@\ell^H : a, :: \mu)$, then have $\cdot \vdash ((x : a, \Delta), ((x@\ell^H : a) :: \mu)) : H\{\ell^H \mapsto \mathbf{FREE}[x : a]\}$ by rule μ-Cons. We also have $\Delta, x : a \vdash S\{\{\ell^H \mapsto \mathbf{FREE}[x : a]\}\} : \Xi; \Psi\{\{\ell^H : \Im(x : a)\}\}$ and $\Delta, x : a; \Psi\{\{\ell^H : \Im(x : a)\}\} \vdash R\{r \mapsto \ell^H\} : \Gamma\{r : \Im(x : a)\}$. By Lemma 10, $\Delta, x : a; S\{\{\ell^H \mapsto \mathbf{FREE}[x : a]\}\} \vdash T$ ok which together with the previous statements gives us $\cdot \vdash m'$ ok.

case $$\frac{R(r) = \ell^H \quad \mathsf{end}(S, \ell^H) = \ell'^H \quad S(\ell'^H) = \mathbf{FREE}[x : a]}{(T, \Delta, S, R, \mathtt{get\_str}\ c, r; I) \longmapsto \mathtt{write}(T, \Delta, S, R, I, c, \ell'^H, \epsilon)}\ \textsc{GetStr} \longmapsto \mathrm{W}$$

By assumption, suffices to show $\Delta; \Psi \vdash (\vec{\ell}^H, r, c)\ \mathtt{writes}\ J$. Since $\vec{\ell}^H$ is empty, we need only know $\Sigma(c) = \vec{a}_2 \longrightarrow a$ where J accepts $\vec{a}_2$ which is true by assumption. We also need $S(\ell'^H) = \mathbf{FREE}[x : a]$ which is true by case.

$$\textbf{case}\quad \frac{R(r) = \ell^H \quad \mathsf{end}(S, \ell^H) = \ell'^H \quad S(\ell'^H) = c\langle \ell_1^H, \ldots, \ell_n^H\rangle}{(T, \Delta, S, R, \mathtt{get\_str}\ c, r; I) \longmapsto \mathtt{read}(T, \Delta, S, R, I, \vec{w})}\ \textsc{GetStr} \longmapsto \mathrm{R}$$

Suffices to show $\vec{\ell}^H$ `reads` J, and in particular $\Delta \vdash M_1 \sqcap M_2 = \sigma$. Since $M_1 = x, M_2 = c\ \vec{x}$ and $x \notin \vec{x}$ (because x `fresh`), the unification succeeds with $x \sqcap c\ \vec{x} = [c\ \vec{x}/x]$.

$$\textbf{case}\quad \frac{\Sigma(c) = \vec{a} \to a}{(T, \Delta, S, R, \mathtt{put\_str}\ c, r; I) \longmapsto \mathtt{write}(T, (x : a, \Delta), S\{\{\ell^H \mapsto \mathbf{FREE}[x : a]\}\}, R\{r \mapsto \ell^H\}, I, c, \ell^H, \epsilon)}\ \textsc{PutStr} \longmapsto$$

Similar to the `write` case of `getstr`. Define $\mu' = (x@\ell^H : a), \mu$, then $\cdot \vdash (((x@\ell^H : a) :: \mu), (\Delta, x : a)) : S\{\{\ell^H \mapsto \mathbf{FREE}[x : a]\}\}$. Now $S(\ell^H) = \mathbf{FREE}[x : a]$ as needed, and by assumption, J accepts $\vec{a}_2$.

$$\textbf{case}\quad \frac{}{(T, \Delta, S, R, \mathtt{put\_tuple}\ r, n; I) \longmapsto \mathtt{twrite}(T, \Delta, S, R, I, r, n, \epsilon)}\ \textsc{PutTuple} \longmapsto$$

Suffice to show $\Delta; \Psi \vdash (\vec{w}, r, n)$ `writes` J where in this case $\vec{w}$ is empty and $J = \vec{\tau} \to \{r : \mathrm{x}[\vec{\tau}]\}$. Since $n = |\tau|$, we have $\Delta; \Psi \vdash (\epsilon, r, n)$ `writes` $\vec{\tau} \to \{r : \mathrm{x}[\vec{\tau}]\}$.

$$\textbf{case}\quad \frac{R(r_s)\ell^H \quad S(\ell^H) = \langle w_1, \ldots, w_i, \ldots, w_n\rangle}{(T, \Delta, S, R, \mathtt{proj}\ r_d, r_s, i; I) \longmapsto (T, \Delta, S, R\{r_d \mapsto w_i\}, I)}\ \textsc{Proj} \longmapsto$$

By inversion and Lemma 1, $R(r_s) = \ell^H$ and $\Delta; \Psi \vdash \ell^H : \mathrm{x}[\vec{\tau}]$ so by Lemma 1, $S(\ell^H) = \langle w_1, \ldots, w_i, \ldots, w_n\rangle$ and $\Delta; \Psi \vdash w_i : \tau_i$. Therefore $\Delta; \Psi \vdash R\{r_d \mapsto w_i\} : \Gamma\{r_d : \tau_i\}$ so $\cdot \vdash m$ ok.

$$\textbf{case}\quad \frac{R(r_s) = w \quad n > 0}{\mathtt{twrite}(T, \Delta, S, R, \mathtt{set\_val}\ r_s; I, r_d, n, \vec{w}) \longmapsto \mathtt{twrite}(T, \Delta, S, R, I, r_d, n - 1, (\vec{w} :: w))}\ \textsc{SetVal} \longmapsto$$

Consider $\vec{\tau}_1, \vec{\tau}_2$ from the derivation $\cdot \vdash (\vec{w}, r, n)$ `writes` J. Observe by assumption $\vec{\tau}_2$ has form $\tau, \vec{\tau}_2'$. Now let $\vec{\tau}_1' = \vec{\tau}_1, \tau$. This gives us $\Delta; \Psi \vdash (\vec{w} :: w) : \vec{\tau}_1'$ and $|\vec{\tau}_2'| = n - 1$ and $\vec{\tau}_1'\vec{\tau}_2' = \vec{\tau}_1\vec{\tau}_2$ so $\Delta; \Psi \vdash ((\vec{w} :: w), r, n - 1)$ `writes` $(\tau_2' \to \{r : \mathrm{x}[\vec{\tau}_1'\vec{\tau}_2']\})$. The rest is by assumption.

$$\textbf{case}\quad \frac{}{\mathtt{twrite}(T, \Delta, S, R, I, r, 0, \vec{w}) \longmapsto (T, \Delta, S\{\{\ell^H \mapsto \langle \vec{\ell}^H\rangle\}\}, R\{r \mapsto \ell^H\}, I)}\ \textsc{TWrite} \longmapsto$$

Since $n = 0$, the instruction typing derivation has form $\Delta; \Gamma \vdash I : \{r : \mathrm{x}[\vec{\tau}]\}$ where $\Delta; \Psi \vdash \vec{\ell}^H : \vec{\tau}$ (from the derivation $\Delta; \Psi \vdash (\vec{\ell}^H, r, 0)$ `writes` J). This derivation must contain $\Delta; \Gamma\{r : \mathrm{x}[\vec{\tau}]\} \vdash I$ ok. We also have $\Delta; \Psi \vdash \langle \vec{w}\rangle : \mathrm{x}[\vec{\tau}]$ so $\Delta \vdash S\{\{\ell^H \mapsto \langle \vec{w}\rangle\}\} : (\Xi; \Psi\{\{\ell^H : \mathrm{x}[\vec{\tau}]\}\})$ and $\Delta; \Psi\{\{\ell^H : \mathrm{x}[\vec{\tau}]\}\} \vdash R\{r \mapsto \ell^H\} : \Gamma\{r : \mathrm{x}[\vec{\tau}]\}$ so $\cdot \vdash m'$ ok.

$$\textbf{case}\quad \frac{}{\mathtt{read}(T, \Delta, S, R, I, \epsilon) \longmapsto (T, \Delta, S, R, I)}\ \textsc{Read} \longmapsto$$

Then $J = c\ \vec{M} \sqcap c\ \vec{M}$ which unifies under the empty substitution, so by assumption, $\Delta; \Gamma \vdash I$ ok so $\cdot \vdash m'$ ok.

$$\textbf{case}\quad \frac{S(\ell'^H) = \mathbf{FREE}[x : a] \quad \ell'^H \notin c\langle \vec{\ell}^H\rangle \quad |\vec{\ell}^H| = \mathtt{arity}(c) \quad \mathtt{update\_trail}(T, x@\ell'^H : a) = T' \quad \mathsf{end}(S, \ell^H) = \ell'^H}{\mathtt{write}(T, \Delta, S, R, I, \ell^H, c, \vec{\ell}^H) \longmapsto (T', [c\ \vec{M}/x]\Delta, S\{\ell^H \mapsto c\langle \vec{\ell}^H\rangle\}, R, [c\ \vec{M}/x]I)}\ \textsc{Write} \longmapsto$$

By Lemma 10, $[M/x]\Delta, S\{\ell'^H \mapsto c\langle \vec{\ell}^H\rangle\} \vdash T'$ ok. Then by assumption, $\Delta; \Psi \vdash \vec{\ell}^H : \Im(\vec{M} : \vec{a})$ and $\Sigma(c) = \vec{a} \to a$ so $\Delta; \Psi \vdash c\langle \vec{\ell}^H\rangle : \Im(c\ \vec{M} : a)$. By assumption and Lemma 12, $S(\ell'^H) = \mathbf{FREE}[x : a]$. The typing derivation for J has the form

$$\frac{\Delta \vdash x \sqcap c\ \vec{M} = \sigma \quad [\sigma]\Delta; [\sigma]\Gamma \vdash [\sigma]I\ \mathsf{ok}}{\Delta; \Gamma \vdash I :_s (x \sqcap c\ \vec{M})}\ \sqcap\sigma$$

By inversion, $\sigma = [c\ \vec{M}/x]$. Therefore $[c\ \vec{M}/x]\Delta; [c\ \vec{M}/x]\Gamma \vdash [c\ \vec{M}/x]I$ ok so it suffices to show $\Delta \vdash S\{\ell^H \mapsto c\langle \vec{\ell}^H\rangle\} : (\Xi; [c\ \vec{M}/x]\Psi)$ which it does by Lemma 9.

$$\frac{R(r) = w}{\textbf{case}\ \mathtt{read}(T, \Delta, S, R, (\mathtt{unify\_var}\ r, x : a.I), \vec{\ell}^H) \longmapsto \mathtt{read}(T, \Delta, S, R, I, (\vec{\ell}^H :: w))}\ \textsc{UnifyVar} \longmapsto \textsc{R}$$

By inversion, $\Gamma(r) = \mathfrak{S}(M : a)$ so by Lemma 23, $\Delta; \Gamma \vdash [M/x]I :_s [M/x]J$ and $\cdot \vdash m'$ ok.

$$\frac{R(r) = \ell'^H \quad \mathtt{unify}(\Delta, S, T, \ell^H, \ell'^H) = (\Delta', S', T')}{\textbf{case}\ \mathtt{read}(T, \Delta, S, R, (\mathtt{unify\_val}\ r, x : a.I), (\ell^H :: \vec{\ell}^H)) \longmapsto \mathtt{read}(T', \Delta', S', R, I, \vec{\ell}^H)}\ \textsc{UnifyVal} \longmapsto \textsc{R}$$

By assumption $\Delta \vdash x \sqcap M = \sigma$ for some $\sigma = [M/x]$. By Lemma 13, $\Delta' \vdash S' : (\Xi; \Psi')$ and $\Delta'; S' \vdash T'$ ok. By Lemma 23, $\Delta; \Gamma \vdash [M/x]I : [M/x]J$. We now also have $\Delta \vdash \vec{\ell}^H\ \mathtt{reads}\ \Pi\vec{x} : \vec{A}.(c\ \vec{M}\ M\ \vec{x}' \sqcap c\ \vec{M}\ M\ \vec{M}'')$ as desired, so $\cdot \vdash m'$ ok.

$$\frac{}{\mathtt{write}(T, \Delta, S, R, (\mathtt{unify\_var}\ r, x : a.I), c, \ell_d^H, \vec{\ell}^H) \longmapsto}\ \textsc{UnifyVar} \longmapsto \textsc{W}$$
$$\textbf{case}\ \mathtt{write}(T, (x : a, \Delta), S\{\{\ell^H \mapsto \mathbf{FREE}[x : a]\}\}, R\{r \mapsto \ell^H\}, I, c, \ell_d^H, (\vec{\ell}^H :: \ell^H))$$

By Lemma 10, $\Delta, x : a; S\{\{\ell^H \mapsto \mathbf{FREE}[x : a]\}\} \vdash T$ ok. Let $\mu' = ((x@\ell^H : a), :: \mu)$. Now $\cdot \vdash ((\Delta, x : a), ((x@\ell^H : a) :: \mu)) : H\{\{\ell^H \mapsto \mathbf{FREE}[x : a]\}\}$. Take $\vec{a}_1, \vec{a}_2$ from the derivation $\Delta; \Psi \vdash (\vec{\ell}^H, \ell_d^H, c)\ \mathtt{writes}\ \Pi\vec{x} : \vec{\tau}.(M_1 \sqcap M_2)$. Note $\vec{a}_2$ has form $a, \vec{a}_2'$. Let $\vec{a}_1' = (\vec{a}_1 :: a')$. Now $(\Delta, x : a); \Psi\{\{\ell^H : \mathbf{FREE}[x : a]\}\} \vdash (\vec{\ell}^H, \ell^H) : \vec{a}_1'$ so $\cdot \vdash m'$ ok.

$$\frac{R(r) = \ell^H}{\textbf{case}\ \mathtt{write}(T, \Delta, S, R, (\mathtt{unify\_val}\ r, x : a.I), c, \ell_d^H, \vec{\ell}^H) \longmapsto \mathtt{write}(T, \Delta, S, R, I, c, \ell_d^H, (\vec{\ell}^H :: \ell^H))}\ \textsc{UnivyVal} \longmapsto \textsc{W}$$

Take $\vec{a}_1, \vec{a}_2$ from the derivation $\Delta; \Psi \vdash (\vec{\ell}^H, \ell_d^H, c)\ \mathtt{writes}\ \Pi\vec{x} : \vec{\tau}.(M_1 \sqcap M_2)$. Note $\vec{a}_2$ has form $a', \vec{a}_2'$. Let $\vec{a}_1' = (\vec{a}_1, a')$. Now $\Delta; \Psi \vdash (\vec{\ell}^H, \ell^H) : \vec{a}_1'$ so $\cdot \vdash m'$ ok. □

6.5 Operands, `mov` and Tail Calls

Now that we have proven TWAM sound, we illustrate its optimization potential by describing our implementation of tail-call optimization (TCO), a common and performance-critical optimization. We take as an example the predicate f:

```
f : t -> prop.
f(X) :- g(X).
```

where the definition of g is irrelevant. In our LF translation, this predicate has one proof term constructor: `F-X:Π X:t.ΠD:g X.f X`. If we compile this program naïvely, it would produce the following code:

Example 6.2 (Before Tail-Call Optimization).

```
f_main ↦ code[X:nat. {r1:𝔖(X), r2: ΠD: f X. ¬{}}(
  λX: nat.
  mov f_tail, r2;
  jmp (g X)
),
  # The following code value should not be necessary
f_tail ↦ code[X:nat, D:g X. {r1:𝔖(X), r2: ΠD: f X. ¬{}}(
  λX: nat, D: g X.
  jmp (r2 (F-X D))
)
```

The problem is that to find a proof of `f X` we must first find a proof `D : g X` and then apply `F-X X D`. Because `F-X` uses `D` we must apply it after `g` has succeeded, which means we must

apply it in g's success continuation. This is a problem: g is supposed to be a tail call, so its success continuation should be the one passed to `f`.

Since LF proofs are completely unnecessary at runtime, our ideal solution would be no-op of sorts: an instruction that allows us to perform simple proof steps in LF, but which can trivially be deleted after typechecking to avoid any runtime cost. This no-op is easily expressed as a special case of the `mov` instruction. The following `mov` instruction takes the success continuation and pre-composes an LF term that converts proofs of `g X` into proofs of `f X` as needed by the continuation.

Example 6.3 (After Tail-Call Optimization)**.**

```
f_main ↦ code[X:t.{r1 :ΠD : f X.¬{}, r2 : ∈(X)}](
  λX:t.
  mov r1 (λ D: g X. r1 (f/X X D));
  jmp (g X)
)
```

7 IMPLEMENTATION

The full source code for our compiler implementation is available from the author upon request, along with a small test suite. The compiler consists of approximately 5000 lines of Standard ML, and has the following phases:

- Parsing and Elaboration: T-Prolog is parsed with `ML-Lex` and `ML-Yacc`, then check T-Prolog types.
- Flattening: Terms with nested constructors are flattened.
- Main Translation: Generates a variant of TWAM where failure continuations are stored inline.
- Hoisting: Lifts failure continuations to the top level.
- Certification: Runs the TWAM typechecker on hoisted code. On failure, signals a compile error.
- Type Erasure: Trivial conversion from TWAM code to SWAM code.
- Rechecking: As a sanity check, type-check the SWAM code.If this fails, signals a compiler error.
- Interpretation: The SWAM code is interpreted, following the operational semantics.

The primary goal of this implementation, which it achieved, was to validate the design of TWAM, especially to show that it is expressive enough to support an interesting source language. For this reason, we intentionally omitted most optimizations except tail-call optimization, which we deemed too essential to ignore. That being said, there are a number of avenues available to build a compiler with more competitive performance:

- Compile to machine code instead of interpreting TWAM code.
- Replace our trivial register allocator with an efficient (e.g. graph-coloring-based) one.
- Implement existing WAM optimizations (e.g. optimized switch statements).
- Investigate the cost of continuation-passing style. Implement optimizations to reduce the number of environments allocated, or develop a stack-based system should that be insufficient.
- Reduce the use of the occurs check by adding a mode system.

Among these, the first two options can be implemented with no changes to the instruction set or type system, and the third can be implemented by adding new instructions without modifying the existing instructions. The final two optimizations are more fundamental in nature, requiring changes to existing instructions or changes that affect the entire type system.

8 CONCLUSION

We have designed and implemented a typed compiler for T-Prolog by first creating a *certifying abstract machine* for logic programs, called the TWAM. Our implementation demonstrates that the TWAM is expressive enough to use as a compilation target for real programs, and that the implementation burden of TWAM is acceptable. This implementation result supports our primary contributions: the development of the TWAM and its metatheory. The metatheory shows that TWAM typechecking suffices to enable certifying compilation in theory, and our compiler shows it in practice.

Our work differs from previous work on Prolog compilation because we are the first to take a typed compilation approach. We have also produced a working compiler with a formal guarantee, whereas previous efforts stopped before implementing a compiler [2, 4, 22, 24]. Several optimizing compilers have been verified in proof assistants [11, 12] and some of them use proof-producing compilation [17], but these do not address logic programming languages. Typed compilation is from our perspective an instance of certifying compilation[18], and proof-passing style specifically is a variant that allows us reason about semantic preservation.

Our type system relies on the logical framework LF [10], and is inspired by other languages with dependent type systems [27], though the languages differ greatly. Our formalisms are inspired by typed assembly languages, but we make major changes to provide stronger guarantees and support logic programming [16].

Future work includes developing a production-quality optimizing compiler and runtime, including any changes to the core TWAM language to enable greater efficiency. We also wish to extend our abstract machine to support logic programming languages with advanced type system features and investigate whether certifying abstract machines can provide equally strong guarantees for non-logic programming languages.

REFERENCES

[1] Hassan Aït-Kaci. 1991. *Warren's abstract machine: A tutorial reconstruction.* MIT Press.

[2] Christoph Beierle and Egon Börger. 1992. Correctness proof for the WAM with types. *Computer Science Logic, LNCS volume 626* (1992), 15–34.

[3] Rose Bohrer and Karl Crary. 2016. A Proof-Producing Verified Prolog Compiler. Tech. Rep. CMU-CS-16-104. Available from first author upon request. (2016). Draft.

[4] Egon Börger and D. Rosenzweig. 1994. The WAM—Definition and Compiler Correctness. *Logic Programming: Formal Methods and Practical Applications* (1994).

[5] Iliano Cervesato. *Proof-Theoretic Foundation of Compilation in Logic Programming Languages* (????), 115–129.

[6] Karl Crary. 2003. Toward a Foundational Typed Assembly Language. *POPL '03* (2003), 198–212. DOI: https://doi.org/10.1145/604131.604149

[7] Karl Crary and Susmit Sarkar. 2008. Foundational Certified Code in the Twelf Metalogical Framework. *ACM Trans. Comput. Logic* 9, 3, Article 16 (June 2008), 26 pages. DOI: https://doi.org/10.1145/1352582.1352584

[8] Karl Crary and Joseph C. Vanderwaart. 2002. An expressive, scalable type theory for certified code. *ICFP* (2002), 191–205.

[9] Conal Elliott and Frank Pfenning. 1991. A Semi-Functional Implementation of a Higher-Order Logic Programming Language. In *Topics in Advanced Language Implementation.* MIT Press, 289–325.

[10] Robert Harper, Furio Honsell, and Gordon Plotkin. 1993. A Framework for Defining Logics. *JACM* (1993).

[11] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. *POPL '14* (2014), 179–191. DOI: https://doi.org/10.1145/2535838.2535841

[12] Xavier Leroy. 2006. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. *POPL '06* (2006).

[13] Guodong Li, Scott Owens, and Konrad Slind. 2007. Structure of a proof-producing compiler for a subset of higher order logic. *ESOP* (2007).

[14] Alberto Martelli and Ugo Montanari. 1982. An Efficient Unification Algorithm. *TPLS* '82, 2 (April 1982), 258–282. DOI: https://doi.org/10.1145/357162.357169

[15] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. 2002. Stack-based Typed Assembly Language. *JFP* '02 (Jan. 2002), 43–88. DOI: https://doi.org/10.1017/S0956796801004178
[16] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From System F to Typed Assembly Language. *TPLS* 21, 3 (May 1999), 527–568. DOI: https://doi.org/10.1145/319301.319345
[17] Magnus O. Myreen and Scott Owens. 2014. Proof-producing translation of higher-order logic into pure and stateful ML. *JFP* '14 (May 2014), 284–315.
[18] George C Necula and Peter Lee. 1998. The Design and Implementation of a Certifying Compiler. *ACM SIGPLAN Notices* 33, 5 (1998), 333–344.
[19] Frank Pfenning. 1989. Elf: A Language for Logic Definition and Verified Metaprogramming. *LICS* (1989), 313–322.
[20] Frank Pfenning. 1991. Unification and Anti-Unification in the Calculus of Constructions. *LICS* (1991), 74–85.
[21] Frank Pfenning and Carsten Schürmann. 1999. System description: Twelf–a meta-logical framework for deductive systems. *CADE* '99 (1999), 202–206.
[22] Cornelia Pusch. 1996. Verification of Compiler Correctness for the WAM. *TPHOLs '96* (1996), 347–362.
[23] J. A. Robinson. 1965. A Machine-Oriented Logic Based on the Resolution Principle. *JACM* '65, 1 (1965), 19. DOI: https://doi.org/10.1145/321250.321253
[24] David M. Russinoff. 1992. A Verified Prolog Compiler for the Warren Abstract Machine. *Journal of Logic Programming* '13 (1992), 367–412.
[25] David HD Warren. 1983. *An abstract Prolog instruction set.* Vol. 309. Artificial Intelligence Center, SRI International Menlo Park, California.
[26] Jan Wielemaker. 2016. SWI-Prolog OpenHub Project Page. https://www.openhub.net/p/swi-prolog. (2016). Accessed: 2016-01-22.
[27] Hongwei Xi and Dana Scott. 1998. Dependent Types in Practical Programming. *POPL '98* (1998), 214–227.
[28] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. *SIGPLAN Not.* 46, 6 (June 2011), 283–294. DOI: https://doi.org/10.1145/1993316.1993532

Received April 2017