TWAM: A Certifying Abstract Machine for Logic Programs

Brandon Bohrer[0000−0001−5201−9895] and Karl Crary[0000−0002−1556−2183]

Carnegie Mellon University, Pittsburgh, PA 15213, USA
{bbohrer,crary}@cs.cmu.edu

Abstract. 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 logic programming language 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 logical signature specified by their type. This metatheorem justifies our design and implementation of a certifying compiler from T-Prolog to TWAM.

1 Introduction

Compiler verification is important because compilers are essential and because compiler bugs are easy to make, yet often difficult to catch. Most work on compiler verification has been done in the setting of imperative or functional programming; little has been done for logic programming.

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. Assuring compiler correctness is of interest for logic programming given the scale of realistic language implementations; for example, SWI-Prolog is estimated at over 600,000 lines of code [27].

Certifying compilation [18] is an approach to verification wherein a compiler outputs a formal proof (in our case, type information) that the compiled program satisfies some desired property. 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.

In most certifying compilation [18] work, an additional disadvantage is that dynamic correctness is not certified, only type and memory safety. In contrast,

we certify that dynamic behavior is sound (but not necessarily complete) w.r.t. to a source semantics given by a signature Σ in the logical framework LF [10]:

Theorem 1(*Soundness*): If query `?-`G. succeeds, there is a proof of G in LF.

This theorem is most meaningful when the source language corresponds closely to proof search in LF. We introduce such a language, called T-Prolog, obtained by removing non-logical features (e.g. cut, negation-as-failure) from Prolog and adding both simple types and an occurs check to rule out infinite terms. Because the semantics of LF are in close harmony with the semantics of T-Prolog, soundness w.r.t LF naturally encompasses (partial) dynamic correctness for T-Prolog. This semantics abstracts away operational details of T-Prolog semantics such as order of execution (and thus termination). While it would certainly be desirable to ensure execution order and termination are compiled correctly, consider common applications of logic programming such as proof/type checking or proof search. Soundness w.r.t. to LF signatures often suffices to state correctness properties for such programs, e.g.: "Every program accepted by the typechecker follows from the typing rules." Especially when we are interested in verifying the source program itself, we are more often interested in verifying soundness than completeness. Since we certify that compilation preserves soundness, any soundness theorems for the source transfer to the compiled code.

The heart of this work is the development of our compilation target, the Typed Warren Abstract Machine (TWAM), a dependently-typed *certifying abstract machine* for logic programs inspired by the Warren Abstract Machine (WAM) [26]. TWAM diverges from WAM in several ways to aid in formalization: (1) we use continuation-passing style for success continuations instead of a stack and (2) we sometimes replace compound instructions (such as those for managing backtracking) with a smaller set of simpler, more orthogonal instructions. As formalized and proved in Section 3, soundness of the TWAM type system says well-typed programs satisfy Theorem 1: well-typed programs are sound proof search procedures for their LF signature. We have implemented a compiler from T-Prolog to TWAM and an interpreter for the TWAM bytecode, which we have tested on a small library. The result is a certifying compiler with a special-purpose proof checker as its trusted core: the TWAM typechecker.

Prerequisites Due to space constraints, we assume familiarity with LF [10] and the WAM. Unfamiliar readers may be interested in our much-extended paper [3], which has a gentler introduction, proofs, full definitions (abridged here), and a simply-typed WAM. Aït-Kaci [1] provides a readable treatment of the WAM.

2 Certifying Compilation in Proof-Passing Style

We briefly demonstrate (Figure 1) the T-Prolog source syntax and the extraction of an LF signature Σ from T-Prolog. We consider addition on the Peano naturals as a running example, i.e., a predicate `plus(X,Y,Z)` that holds when $X+Y=Z$. We write `0` and `1+` for the Peano natural constructors. A T-Prolog program consists of standard Prolog syntax plus type annotations. Throughout the paper,

we write vectors in bold, e.g.; $\boldsymbol{a}$ below. Throughout, a is a simple (inductive) type in T-Prolog while A is an LF type family.

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

```
nat:type.                          nat:type.
0:nat.                             0:nat.
1:nat +→ nat.                      1+:nat → nat.
plus:nat → nat → nat → prop.       Plus:nat → nat → nat → type.
plus(0,X,X).                       Plus-Z:ΠX:nat. Plus 0 X X.
plus(1+(X),Y,1+(Z)) :-             Plus-S:ΠX:nat. ΠY:nat. ΠZ:nat.
  plus(X,Y,Z).                       ΠD:Plus X Y Z.
                                       Plus (1+ X) Y (1+ Z).
```

Fig. 1. Example T-Prolog program and LF signature

The TWAM certification approach can be summed up in a 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 output programs do not segfault. 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 (Theorem 1).

A TWAM program must contain enough information that the TWAM type-checker can ensure all terminating runs of the program satisfy Theorem 1. We achive this by statically ensuring that whenever *each* proof search procedure p returns, the corresponding predicate P will have a proof in LF. This amounts to (1) annotating each return point with the corresponding LF proof term and (2) reasoning statically about constraints on T-Prolog terms with dependent *singleton types* $\mathfrak{S}(M : a)$ (`S(M:a)` in ASCII) containing exactly the values that represent some LF term M of simple type a. Singleton typing information is needed to typecheck almost any LF proof term. For example, an application of `Plus-Z` only checks if we statically know the first argument is 0 and that the second and third arguments are equal, all of which are learned during unification.

This *proof-passing* style of programming is a defining feature of the TWAM type system. It is worth noting that these proofs never need to inspected at runtime and thus can be (and in our implementation, are) *erased* before execution. Thus certification adds no runtime overhead.

Keywords `type` and `prop` are distinguished solely for aesthetic purposes: in the theory, they are identical.

3 The Typed WAM (TWAM)

Our core theoretical contributions are the design and metatheory of a certifying abstract machine, the TWAM. In this section we present the dependently-typed TWAM and prove (Section 3.4) that it provides partial correctness guarantees.

3.1 Syntax

instructions	$\iota ::= \texttt{succeed}[M : A] \mid \texttt{mov}\ r_d, op \mid \texttt{jmp}\ op \mid \texttt{put\_str}\ c, r$
	$\mid \texttt{unify\_var}\ r, x : a \mid \texttt{unify\_val}\ r \mid \texttt{get\_val}\ r_1, r_2$
	$\mid \texttt{put\_var}\ r, x : a \mid \texttt{close}\ r_d, r_e, (\ell^C\ \boldsymbol{M}) \mid \texttt{push\_bt}\ r_e, (\ell^C\ \boldsymbol{M})$
	$\mid \texttt{put\_tuple}\ r, n \mid \texttt{set\_val}\ r \mid \texttt{proj}\ r_d, r_s, i$
instruction sequences	$I ::= \cdot \mid \iota; I$
trails,frames,trails	$T ::= \langle\rangle \mid (tf :: T) \quad tf ::= (w_{\text{code}}, w_{\text{env}}, tr) \quad tr ::= \langle\rangle \mid (x{:}a@\ell^H) :: tr$
code sections,heaps	$C ::= \{\ell_1^C \mapsto v_1^C, \ldots, \ell_n^C \mapsto v_n^C\}\ H ::= \{\ell_1^H \mapsto v_1^H, \ldots, \ell_n^H \mapsto v_n^H\}$
stores	$S ::= (C, H)$
heap values	$v^H ::= \mathbf{FREE}[x : a] \mid \mathbf{BOUND}\ \ell^H \mid c\langle \ell_1^H, \ldots, \ell_n^H \rangle$
	$\mid \texttt{close}(w_{\text{code}}, w_{\text{env}}) \mid (\boldsymbol{w})$
code values	$v^C ::= \texttt{code}[\Pi \boldsymbol{x} : \boldsymbol{A}.\Gamma](I)$
word values	$w ::= \ell^C \mid \ell^H \mid w\ M \mid \lambda x : A.w$
register files	$R ::= \{\texttt{r0} \mapsto w_0, \ldots, \texttt{rn} \mapsto w_n\}$
machines	$m ::= (\Delta, T, S, R, I) \mid \texttt{write}(\Delta, T, S, R, I, c, \ell, \boldsymbol{\ell})$
	$\mid \texttt{read}(\Delta, T, S, R, I, \boldsymbol{\ell}) \mid \texttt{twrite}(\Delta, T, S, R, I, r, n, \boldsymbol{w})$
value types	$\tau ::= \mathfrak{S}(M : a) \mid \Pi \boldsymbol{x} : \boldsymbol{A}.\neg\Gamma \mid \texttt{x}[\boldsymbol{\tau}]$
register file types	$\Gamma ::= \{\texttt{r0}\colon \tau_0, \ldots, \texttt{rn}\colon \tau_n\}$
heap, code sec. types	$\Psi ::= \{\ell_1^H : \tau_1, \ldots, \ell_n^H : \tau_n\} \quad \Xi ::= \{\ell_1^C : \tau_1, \ldots, \ell_n^C : \tau_n\}$
spine types	$J ::= \Gamma \mid \Pi x : a \to J \quad J_t ::= \boldsymbol{a} \to \{r_d : \tau\}$
signatures	$\Sigma ::= \cdot \mid \Sigma, c\colon a_1 \to \cdots \to a_n$

Fig. 2. TWAM instructions, machine state dynamics, machine states, typing constructs

The text of a TWAM program is formalized as a *code section* C mapping an identifier ℓ^C for each basic block to a *code value* v^C. The code values are all *code literals* $\texttt{code}[\Pi \boldsymbol{x}{:}\boldsymbol{A}.\Gamma](I)$ where I is a basic block (instruction sequence) and $\Pi \boldsymbol{x}{:}\boldsymbol{A}.\Gamma$ is a *register file type* (with $\boldsymbol{x}, \boldsymbol{A}$ possibly empty) specifying LF parameters and expected initial register types for I. As in LF, Π (`Pi` in ASCII) is a dependent function type.

We present the code section for `plus` as an example, consisting of two code values: `plus-zero/3` and `plus-succ/3`. For completeness we include a code value `query/0` for the example query `plus(X, 0, 1+(0))` and a code value for the top-level success continuation `init-cont`. Like all TWAM code, it is written in

continuation passing style: code values never return, but rather return control to the caller by jumping to a success continuation passed in through a register. Because functions never return, the type of a continuation is fully described by the types its arguments: LF terms and registers, written $\Pi \boldsymbol{x}{:}\boldsymbol{A} \ldots$ and $\neg\Gamma$ (or `!G` in ASCII) respectively, where Γ is a register file type.

Example 31 (Implementing `plus`)**.**

```
# Entry point to plus, implements
# case plus(0,X,X) and tries
# plus-succ/3 on failure
plus-zero/3: code[Pi X,Y,Z:nat.
  {A1:S(X),A2:S(Y),A3:S(Z),
   ret:(Pi _:(Plus X Y Z). !{})}](
  put_tuple X1, 4;
    set_val A1;
    set_val A2;
    set_val A3;
    set_val ret;
  push_bt X1, (plus-succ/3 X Y Z);
  get_str A1, 0;
  get_val A2, A3;
  jmp (ret (Plus-Z Y));)
```

```
# plus(1+(X), Y, 1+(Z))
#    :- plus(X,Y,Z).
plus-succ/3: code[Pi X,Y,Z:nat.
  {env: x[S(X),S(Y),S(Z),
     (Pi_:(Plus X Y Z).!{})]}](
  proj A1, env, 1;
  proj A2, env, 2;
  proj A3, env, 3;
  proj ret, env, 4;
  get_str A1, 1+;
  #Set arg 1 of rec. call to X-1
    unify_var A1, XX:nat;
  get_str A3, 1+;
  #Set arg 3 of rec. call to Z-1
    unify_var A3, ZZ:nat;
  #tail-call optimization: add
  #Plus-S constructor when called
  mov ret, (lam D:plus XX Y ZZ.
    ret (Plus-S XX Y ZZ D));
  jmp (plus-zero/3 XX Y ZZ))
```

Example 32 (Calling `plus`)**.**

```
init-cont/0:code[Pi X:nat.
  Pi D:(Plus X 0 (1+ 0))](
 succeed[D:Plus X 0 (1+ 0)])
```

```
# plus(X, 0, 1+(0))
query/0:code[{}](
  put_var A1, X:nat;
  put_tuple X1, 0;
  close ret, X1, (init-cont/0 X);
  put_str A2, 0;
  put_str A3, 1+;
    unify_val A2;
  jmp(plus-zero/3 X 0 (1+ 0)))
```

The query entry point is `query/0`. The `plus` entry point is `plus-zero/3`, which is responsible for implementing the base case $r_1 = 0$. Its type annotation states that the arguments are natural numbers passed in arguments A_1 through A_3. The success continuation (return address) is passed in through `ret`, but may only be invoked once `Plus X Y Z` is proved.

The instructions themselves are similar to the standard WAM instructions. `plus-zero/3` is implemented by attempting to unify A_1 with `0` and A_2 with A_3. If the `plus-zero/3` case succeeds, we return to the address stored in `ret`,

proving `Plus X Y Z` in LF with the `Plus-Z` rule. If the case fails, we backtrack to `plus-succ/3` to try the `Plus-S` case. `plus_succ/3` in turn makes a recursive call to `plus-zero/3` to prove the subgoal $XX + Y = ZZ$, where `XX` and `ZZ` are the predecessors of `X` and `Z`. The `mov` instruction implements proof-passing for tail-calls. Dynamically speaking, we should not need to define a new success continuation because we are making a tail call. However, while `Plus XX Y ZZ` implies `Plus X Y Z`, their proof terms are not the same: proving `Plus X Y Z` requires an extra application of `Plus-S`. This `mov` instruction simply says to apply `Plus-S` (statically) before invoking `ret`, and can be erased before execution.

Machines As shown in Figure 2, the state of a TWAM program is formalized as a tuple $m = (\Delta, T, S, R, I)$ (or a special machine states `read` or`write`: see, e.g., Section 3.2). Here T is the *trail*, the data structure that implements backtracking. The trail consists of a list of *trail frames* (tf), each of which contains a failure continuation (address and environment) and a *trace* (tr), which lists any bound variables which would have to be made free to recover the state in which the failure continuation should be run. In WAM terminology, each frame implements one choice point. The *store* $S = (H : \Psi, C : \Xi)$ contains the heap and code section, $R : \Gamma$ contains the registers and I represents the program counter as a list of all remaining instructions in the current basic block. Typical register names are A_i for arguments, X_i for temporaries, `ret` for success continuations, and `env` for closures being unpacked. Δ contains the free variables of H; it is used primarily in Section 3.4. The *heap* H contains the T-Prolog terms. Heap value $\mathbf{FREE}[x : a]$ is a free variable x of type a and $c\langle \ell_1, \ldots, \ell_n \rangle$ is a *structure*, i.e. a *functor* (cf. constructor in LF) c applied to argument vector $\langle \ell_1, \ldots, \ell_n \rangle$. As in WAM, the heap is in disjoint-set style, i.e. all free variables are distinct and pointers $\mathbf{BOUND}\ \ell$ are introduced when unifying variables; $\mathbf{BOUND}\ \ell$ and ℓ represent the same LF term. TWAM heaps are acyclic, as ensured by an occurs check. The heap also contains success continuation closures $\texttt{close}(w_{\text{code}}, w_{\text{env}})$ and n-ary tuples $(\boldsymbol{w})$ (used for closure environments), which do not correspond to T-Prolog terms.

3.2 Operational Semantics

We give the operational semantics by example . Due to space constraints, see the extended paper [3] for formal small-step semantics (judgements $m \longmapsto^* m'$ and m `done`). Those judgments which will appear in the metatheory are named in this section. We give an evaluation trace of the query `?- plus(X,0,1+(0))`. 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 uses special execution modes *read* and *write* to destruct or construct sequences of arguments to a functor (we dub this sequence a *spine*). When the program enters read mode, we annotate that line with the list ℓs of arguments being read, and when the program enters write mode we annotate it with the constructor c being applied, the destination location ℓ and the argument locations ℓs. The final instruction of a write-mode spine can be seen as two evaluation steps (separately by semicolons below), one of which constructs the last argument of the constructor and one of

which combines the arguments into a term. We write $H\{\{\ell^H \mapsto v^H\}\}$ for heap H extended with new location ℓ^H containing v^H, or $H\{\ell^H \mapsto v^H\}$ for updating an existing location. $R\{r \mapsto w\}$ is analogous. Updates $H\{\ell^H \mapsto v^H\}$ are only guaranteed acyclic when the occurs check passes. Below, all occurs checks pass, and are omitted for brevity.

```
Code:                  Outcome:
  query/0 |-> code [{}](
1   put_var A1, X:nat;  H<-H{{l1->FREE[X:nat]}}, R<-R{A1->l1};
2   put_tuple X1, 0;    H<-H{{l2-> ()}}, R<-R{X1->l2};
3   close ret, X1, (init-cont/0 X);
                        H<-H{{l3->close(init-cont/0 X, l2)}}, R<-R{ret->l3};
4   put_str A2, 0;      H<-H{{l4->0}}, R<-R{A2->l4}
5   put_str A3, 1+;     H<-H{{l5->FREE[_:nat]}}, R<-R{A3->l5} c=1+; l=l5,ls=<>
6     unify_val A2;     ls <- <l4>; H<-H{l5 -> 1+ <l4>}
7   jmp plus-zero/3 ..  I <- (C(plus-zero/3) X 0 (1+ 0))

plus-zero/3: code[Pi X,Y,Z:nat.
  {A1:S(X),A2:S(Y),A3:S(Z),ret:(Pi _:(Plus X Y Z). !{})}](
8  put_tuple X1, 4;       ls=<>, n=4
9     set_val A1;         ls=<l1>
10    set_val A2;         ls=<l1,l4>
11    set_val A3;         ls=<l1,l4,l5>
12    set_val ret;        ls=<l1,l4,l5,l3>;
                          H<-H{{l6->(l1,l4,l5,l3)}}, R<-R{X1->l6}
13  push_bt X1, (plus-succ/3 X Y Z);  T<-(plus-succ/3 X Y Z, l6, <>)::<>
14  get_str A1, 0;        WRITE: H<-H{l1->0},
                          T<-(plus-succ/3 X Y Z, l6, <l1>)::<>
15  get_val A2, A3;  BACKTRACK: T<-<>, I<-plus-succ/3 .., H<-H{l1->FREE[X:nat]}

plus-succ/3 |-> code[Pi X,Y,Z:nat.
  {env: x[S(X),S(Y),S(Z),(Pi_:(Plus X Y Z).!{})]}](
16  proj A1, env, 1;          R<-R{A1->l1}
17  proj A2, env, 2;          R<-R{A2->l4}
18  proj A3, env, 3;          R<-R{A3->l5}
19  proj ret, env, 4;         R<-R{ret->l3}
20  get_str A1, 1+;           WRITE: c=1+, l=l1, ls=<>
21    unify_var A1, XX:nat;   H<-H{{l7->FREE[XX:nat]}},R<-R{A1->l7},ls=<l7>
                              H<-H{l1-> 1+ <l7>}
22  get_str A3, 1+;           READ: ls=<l4>
23    unify_var A3, ZZ:nat;   R<-R{A3->l4}
24  mov ret, (lam D:plus XX Y ZZ. ret (Plus-S XX Y ZZ D));
    R<-R{ret->(lam D:plus XX Y ZZ. l3 (Plus-S XX Y ZZ D))} (NO-OP)
25  jmp (plus-zero/3 XX Y ZZ)); I<- C(plus-zero/3) XX Y ZZ

plus-zero/3 |-> code[Pi X,Y,Z:nat.
  {A1:S(X),A2:S(Y),A3:S(Z),ret:(Pi _:(Plus X Y Z). !{})}](
26  put_tuple X1, 4;      ls=<>, n=4
27    set_val A1;         ls=<l7>
28    set_val A2;         ls=<l7,l4>
```

```
29    set_val A3;          ls=<l7,l4,l4>
30    set_val ret;         ls=<l7,l4,l4,lam .. l3>;
                           H<-H{{l8->(l7,l4,l4,lam .. l3)}}, R<-R{X1->l8}
31  push_bt X1, (plus-succ/3 X Y Z);  T<-(plus-succ/3 X Y Z, l8, <>)::<>
32  get_str A1, 0;         READ ls=<>, l=l7; H<-H{l7->0}
33  get_val A2, A3;        no change, R(A2) = R(A3)
34  jmp (ret (Plus-Z Y)); I <- C(R(r4)) (Plus-Z Y)
            = C(success) 0 Y Z
                  (Plus-S 0 Y Z (Plus-Z Y)))

35  init-cont/0:code[Pi X,Y,Z:nat.Pi D:(Plus X Y Z)](succeed[D:(Plus X Y Z)];)
```

All top-level queries follow the same pattern of constructing arguments, setting a success continuation, then invoking a search procedure. Line 1 constructs a free variable. Line 2 creates an empty environment tuple which is used to create a success continuation on Line 3. This means if proof search succeeds we will return to `init-cont/0`, which immediately ends the program in success. Line 4 is a 1-instruction write-mode spine. First we allocate a free variable at ℓ_4 to store the term, then because we have finished the spine, we bind the variable to `0`. Lines 5-6 are a write spine constructing `1+(0)`. Because A_2 already contains `0`, we can eliminate a common subexpression, reusing it for `1+(0)`. Line 7 invokes the main `Plus` proof search.

Lines 8-12 pack the environment in a tuple. Line 13 creates a trail frame which executes `plus-succ/3` if `plus-zero/3` fails. Its trace is initially empty: from this point, the trace will be updated any time we bind a free variable. Line 14 dynamically checks A_1, observes it is free and thus enters write mode. On line 14 we also bind A_1 to `0` and add it to the trace. Note that this is the first time we add a variable to the trace because we only do so when trail contains at least one frame. The trace logic is formalized in a judgement update_trail. When the trail is empty, backtracking would fail anyway, so there is no need to track variable binding.

Line 15 tries and fails to unify (judgement unify) the contents of A_2 and A_3, so it backtracks to `plus-succ/3` (judgement backtrack).

Backtracking consists of updating the instruction pointer, setting all trail locations to free variables, and loading an enviroment. The `plus-succ/3` case proceeds without trouble: the first `get_str` enters write mode because A_1 is free, but the second enters read mode because A_3 is not free. On Line 26 we enter the `0` case of `plus` with arguments $A_1 = A_2 = A_3 = \texttt{0}$. All instructions succeed, so we reach Line 34 which jumps to line 35 and reports success.

3.3 Statics

This sections presents the TWAM type system. The main judgement $\Delta; \Gamma \vdash I$ ok says instruction sequent I is well-typed. A code section is well-typed if every block is well-typed. The system contains a number of auxilliary judgments, which will be introduced as needed. Most of type-checking is independent of which query we make, thus the query is not mentioned explictly in the judgement

$\Delta; \Gamma \vdash I$ ok. The certifying compiler can check that the output TWAM program makes the same query as the source program by comparing the query with the type annotation on the unique `succeed` instruction in the inital continuation. Below, the notation $\Psi\{\ell\colon \tau\}$ denotes the heap type Ψ with the type of ℓ replaced by τ whereas $\Psi\{\{\ell\colon v\}\}$ denotes Ψ extended with a fresh location ℓ of type τ.

Success. We wish to prove that a program only succeeds if a proof M the desired query A exists in LF. We require exactly that in the typing rule:

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

The `succeed` rule is simple, but deceptively so: the challenge of certifying compilation for TWAM is how to satisify the premiss of this rule. The proof-passing approach says we satisfy this premiss by threading LF terms through every predicate: by the time we reach the `succeed` instruction, the proof will have already been constructed.

Proof-passing. The `jmp` instruction is used to invoke and return from basic blocks. When returning from a basic block, it passes an LF proof term to the success continuation, showing that the corresponding LF predicate has a proof. These LF proof terms are part of the `jmp` instruction's *operand* op:

$$\frac{\Delta; \Gamma \vdash op : \neg\Gamma' \quad \Delta \vdash \Gamma' \leq \Gamma}{\Delta; \Gamma \vdash_{\Sigma;\Xi} \texttt{jmp}\ op, I\ \texttt{ok}}\ \textsc{Jmp}$$

Here $\Delta \vdash \Gamma' \leq G$ means every register of Γ' appears in Γ with the same type.

The *operands* consist of registers, locations and LF terms:

operands $op ::= \ell \mid r \mid op\ M \mid \lambda x\colon A.op$

Operand typechecking is written $\Delta; \Gamma \vdash op : \tau$ and employs standard rules for checking LF terms. The `mov` instruction is nearly standard. It supports arbitrary operands, which are used in our implementation to support tail-call optimization, as seen in Line 24 of the execution trace.

$$\frac{\Delta; \Gamma \vdash op\colon \tau \quad \Delta; \Gamma\{r_d\colon \tau\} \vdash I\ \texttt{ok}}{\Delta; \Gamma \vdash \texttt{mov}\ r_d, op; I\ \texttt{ok}}\ \textsc{Mov}$$

Continuation-passing. Closures are created explicitly with the `close` instruction: `close` $r_d, r_e, \ell^C \boldsymbol{M}$ constructs a closure in r_d which, when invoked, executes the instructions at ℓ^C using LF arguments $\boldsymbol{M}$ and environment r_e. The environment is an arbitrary value which is passed to $\ell^C\ \boldsymbol{M}$ in the register `env`. The argument $(\ell^C\ \boldsymbol{M})$ is an operand, syntactically restricted to be a location applied to arguments.

$$\frac{\Gamma(r_s) = \tau \quad \Delta; \Gamma\{r_d\colon \Pi \boldsymbol{x}\colon \boldsymbol{A}.\neg\Gamma'\} \vdash I\ \texttt{ok} \quad \Delta; \Gamma \vdash (\ell^C\ \boldsymbol{M})\colon \Pi \boldsymbol{x}\colon\ (\boldsymbol{A}.\neg\Gamma'\{\texttt{env}\colon \tau\})}{\Delta; \Gamma \vdash \texttt{close}\ r_d, r_s, (\ell^C\ \boldsymbol{M}); I\ \texttt{ok}}\ \textsc{Close}$$

Trail frames are similar, except they are stored in the trail instead of a register:

$$\frac{\Delta;\Gamma \vdash I\ \mathtt{ok} \quad \Gamma(r_e) = \tau \quad \Delta;\Gamma \vdash (\ell^C\ \boldsymbol{M}) : \neg\{\mathtt{env} : \tau\}}{\Delta;\Gamma \vdash \mathtt{push\_bt}\ r_e, (\ell^C\ \boldsymbol{M}); I\ \mathtt{ok}}\ \text{BT}$$

Singleton Types. The PUTVAR rule introduces an LF variable x of simple type a, corresponding to a TWAM unification variable. Statically, the LF variable is added to Δ. Dynamically, the TWAM variable is stored in r, so statically we have $r : \mathfrak{S}(x : a)$, i.e., r contains a representation of variable x.

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

Singleton typing knowledge is then exploited in type-checking LF proof terms.

Unification. However, `put_var` alone does not provide nearly enough constraints to check most terms. Almost every LF term needs to exploit equality constraints learned through unification. To this end, we introduce a *static* notion of unification $M_1 \sqcap M_2$, allowing us to integrate unification reasoning into our type system and thus into LF proofs. We separate unification into a judgement $\Delta \vdash M_1 \sqcap M_2 = \sigma$ which computes the most general unifier of M_1 and M_2 (or $\bot$ if no unifier exists) and capture-avoiding substitution $[\sigma]\Delta$. We also introduce notation $[\![\sigma]\!]\Delta$ standing for $[\sigma]\Delta$ with the bound variables of σ removed, since unification often removes free variables which might located arbitrarily within Δ. All unification in T-Prolog is first-order, for which algorithms are well-known [23]. One such algorithm is given by the inference rules in the extended paper [3].

The `get_val` instruction unifies its arguments. If no unifier exists, `get_val` vacuously typechecks: we know statically that unification will fail at runtime and, e.g., backtrack instead of executing I. Indeed, this is one of the greatest subtleties of the TWAM type system: all unification performed in the type system is *hypothetical*. At type-checking time we cannot know what arguments a function will ultimately receive, so we treat all arguments as free variables. The great trick (and key to the soundness proofs) is that this does not disturb the typical preservation of typing under substitution. For example, after substituting concrete arguments at runtime, the result will still typecheck even if unification fails, because failing unifications typecheck vacuously.

$$\frac{\begin{array}{c}\Delta \vdash M_1 \sqcap M_2 = \bot \\ \Gamma(r_1) = \mathfrak{S}(M_1 : a) \quad \Gamma(r_2) = \mathfrak{S}(M_2 : a)\end{array}}{\Delta;\Gamma \vdash \mathtt{get\_val}\ r_1, r_2; I\ \mathtt{ok}}\ \textsc{Getval-}\bot$$

$$\frac{\begin{array}{c}\Gamma(r_1) = \mathfrak{S}(M_1 : a) \quad \Gamma(r_2) = \mathfrak{S}(M_2 : a) \\ \Delta \vdash M_1 \sqcap M_2 = \sigma \quad [\![\sigma]\!]\Delta; [\sigma]\Gamma \vdash [\sigma] I\ \mathtt{ok}\end{array}}{\Delta;\Gamma \vdash \mathtt{get\_val}\ r_1, r_2; I\ \mathtt{ok}}\ \textsc{Getval}$$

Tuples and Simple Spines Tuples are similar to structures, except they cannot be unified, may contain closures, and do not have read spines. The `proj` instruction accesses arbitrary tuple elements i:

$$\frac{\Gamma(r_s) = \mathrm{x}[\boldsymbol{\tau}] \quad \Gamma\{r_d\colon \tau_i\} \vdash I\ \mathsf{ok} \quad (\text{where } i \leq |\boldsymbol{\tau}|)}{\Delta; \Gamma \vdash \texttt{proj}\ r_d, r_s, i; I\ \mathsf{ok}}\ \textsc{Proj}$$

New tuple creation is started by `put_tuple`. Elements are populated by a *tuple spine* containing `set_val` instructions. We check the spine using an auxilliary typing judgement $\Delta; \Gamma \vdash I{:}J_t$ where J_t is a *tuple spine* type of form $\boldsymbol{\tau_2} \to \{r_d{:}\mathrm{x}[\boldsymbol{\tau_1}\boldsymbol{\tau_2}]\}$. A tuple spine type encodes both the expected types of all remaining arguments ($\boldsymbol{\tau_2}$) and a postcondition: when the spine completes, register r_d will have type $\mathrm{x}[\boldsymbol{\tau_1}\boldsymbol{\tau_2}]$. The typing rules check each `set_val` in sequence, then return to the standard typing mode $\Delta; \Gamma \vdash I\ \mathsf{ok}$ when the spine completes.

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

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

Dependent Spines. While the `get_val` instruction demonstrates the essence of unification, much unification in TWAM (as in WAM) happens in special-purpose *spines* that create or destruct sequences of functor arguments. Because spinal instructions are already subtle, the resulting typing rules are as well.

We introduce an auxiliary judgement $\Gamma \vdash I\colon J$ and dependent *functor spine types* J. As above, they encode arguments and a postcondition, but here the postcondition is the unification of two terms, and the arguments are dependent.

The base case is $J \equiv (M_1 \sqcap M_2)$, meaning that LF terms M_1 and M_2 will be unified if the spine succeeds. When J has form $\Pi x{:}a.J'$, the first instruction of I must be a spinal instruction that handles a functor argument of type a (recall that the same instructions are used for both read and write mode, as we often do not know statically which mode will be used). The type J' describes the type of the remaining instructions in the spine, and may mention x. The spinal instruction `unify_var` unifies the argument with a fresh variable, while `unify_val` unifies the argument with an existing variable.

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

$$\frac{\Delta, x\colon a; \Gamma\{r\colon \mathfrak{S}(x\colon a)\} \vdash I\colon J}{\Delta; \Gamma \vdash \texttt{unify\_var}\ r, x\colon a; I\colon \Pi x\colon a.J}\ \textsc{Unifyvar}$$

The instruction `get_str` unifies its argument with a term $c\langle X_1, \ldots, X_n\rangle$ by executing a spine as described above. The `put_str` instruction starts a spine

that constructs a new structure.

$$\frac{\Sigma(c) = \boldsymbol{a} \to a \quad \Gamma(r) = \mathfrak{S}(M\colon a) \quad \Delta; \Gamma \vdash I\colon (\Pi \boldsymbol{x}\colon \boldsymbol{a}.(M \sqcap c\ \boldsymbol{x}))}{\Delta; \Gamma \vdash \texttt{get\_str}\ c, r; I\ \mathsf{ok}}\ \textsc{Getstr}$$

$$\frac{\Sigma(c) = \boldsymbol{a} \to a \quad \Delta, x\colon a; \Gamma\{r\colon \mathfrak{S}(x\colon a)\} \vdash I\colon (\Pi \boldsymbol{x}\colon \boldsymbol{a}.(x \sqcap c\ \boldsymbol{x}))}{\Delta; \Gamma \vdash \texttt{put\_str}\ c, r; I\ \mathsf{ok}}\ \textsc{Putstr}$$

This completes the typechecking of TWAM instructions.

Machine Invariants. Having completed instruction checking, we prepare for the metatheory by considering the invariants on validity of machine states, which are quite non-trivial. Consider first the invariant for non-spinal machines:

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

Recall machines include a context Δ containing the free T-Prolog variables of the heap H. We can[1] identify variables of Δ with heap locations, trivially ensuring free variables are represented uniquely. Premisses $\Delta \vdash C{:}\Xi$ and $\Delta; \Gamma \vdash I\ \mathsf{ok}$ and $\Gamma; \Psi \vdash R : \Gamma$ simply say the code section, current basic block, and register file typecheck.

Premiss $\Delta \vdash H{:}\Psi$ says all heap values obey their types and that the heap is acyclic. The encoding of acyclic heaps is subtle: while both the heap H and its type Ψ are unordered, the typing derivation is ordered. The rule for non-empty heaps $H\{\{\ell^H \mapsto v^H\}\}$ says that the new value v may refer only to values that appear earlier in the ordering:

$$\frac{\Delta \vdash H\colon \Psi \quad \Delta; \Psi \vdash v^H\colon \tau \quad \ell^H \notin \mathit{Dom}(H)}{\Delta \vdash H\{\{\ell^H \mapsto v^H\}\}\colon \Psi\{\{\ell^H\colon \tau\}\}}$$

Thus, the derivation exhibits a topological ordering of the heap, proving that it is acyclic. Section 3.4 shows this invariant is maintained because we only bind variables when the occurs check passes. The code section has no ordering constraint: even simple functions like `Plus` have mutually recursive basic blocks.

Heap values for T-Prolog terms have singleton types:

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

$$\frac{\Sigma(c) = \boldsymbol{a} \to a \quad \Delta; \Psi \vdash \ell_i^H\colon \mathfrak{S}(M_i\colon a_i)}{\Delta; \Psi \vdash_{\Sigma;\Xi} c\langle \ell_1^H, \ldots, \ell_n^H\rangle\colon \mathfrak{S}(c\ \boldsymbol{M}\colon a)}$$

[1] While this approach is preferable for the proofs, it is quite unreadable, so we used readable names in our presentation of the example instead.

Premiss $\Delta; C; H \vdash T\ \mathsf{ok}$ says the trail is well-typed. The empty trail $\langle\rangle$ checks trivially. A non-empty trail is well-typed if after *unwinding* the trace tr (i.e. making the traced variables free again) we will be in a well-typed state.

$$\frac{\mathsf{unwind}(\Delta, H, t) = (\Delta', H') \quad \Delta; (C, H') \vdash T\ \mathsf{ok} \quad \Delta \vdash H' : \Psi' \quad \Psi' \vdash w_{env} : \tau \quad \Delta; \Psi' \vdash \ell^C\ \boldsymbol{M} : \neg\{\mathtt{env} : \tau\}}{\Delta; C; H \vdash_{\Sigma;\Xi} (\ell^C\ \boldsymbol{M}, w_{env}, tr) :: T\ \mathsf{ok}}\ \textsc{Trail-Cons}$$

This completes the invariants for non-spinal machines.

The typing invariants for spinal machines have additional premisses $\Delta; \Psi \vdash \boldsymbol{\ell}\ \mathtt{reads}\ \Pi \boldsymbol{x} : \boldsymbol{A}.(c\ \boldsymbol{M}\ \boldsymbol{M'} \sqcap c\ \boldsymbol{M}\ \boldsymbol{x})$ (for read spines) or $\Delta; \Psi \vdash (\boldsymbol{\ell}^H, \ell^H, c)\ \mathtt{writes}\ \Pi \boldsymbol{x} : \boldsymbol{a}_2.x' \sqcap c\ \boldsymbol{M}\ \boldsymbol{x}$ (for write spines). Syntactically, these are among the most complex rules in the entire system. Nonetheless, their high-level goals are natural. For a read spine, the types $\boldsymbol{a}$ expected by the spine type must agree with the types of remaining arguments $\boldsymbol{\ell}$. For a write spine, the types of all values written so far must agree with the functor arguments and the destination must agree with functor result. Naturally, the yet-unwritten arguments must also agree with the functor type, but that is already ensured by $\Delta; \Gamma \vdash I{:}J$.

$$\frac{\Delta; \Psi \vdash \boldsymbol{\ell} : \mathfrak{S}(\boldsymbol{M'} : \boldsymbol{a})}{\Delta; \Psi \vdash \boldsymbol{\ell}\ \mathtt{reads}\ \Pi \boldsymbol{x} : \boldsymbol{a}.(c\ \boldsymbol{M}\ \boldsymbol{M'} \sqcap c\ \boldsymbol{M}\ \boldsymbol{x})}$$

$$\frac{\Delta \vdash C : \Xi \quad \Delta \vdash H : \Psi \quad \Delta; \Gamma \vdash I : J \quad \Delta; \Psi \vdash \boldsymbol{\ell}\ \mathtt{reads}\ J \quad \Delta \vdash T\ \mathsf{ok} \quad \Delta; \Psi \vdash R : \Gamma}{\cdot \vdash \mathtt{read}(\Delta, T, C, H, R, I, \boldsymbol{\ell})\ \mathsf{ok}}$$

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

$$\frac{\Delta \vdash C : \Xi \quad \Delta \vdash H : \Psi \quad \Delta; \Gamma \vdash I : J \quad \Delta; \Psi \vdash (\boldsymbol{\ell}^H, \ell, c)\ \mathtt{writes}\ J \quad \Delta \vdash T\ \mathsf{ok} \quad \Delta; \Psi \vdash R : \Gamma}{\cdot \vdash \mathtt{write}(\Delta, T, C, H, R, I, c, \ell^H, \boldsymbol{\ell})\ \mathsf{ok}}$$

The case for tuple spines is similar to the write case.

3.4 Metatheory

Proofs of metatheorems are in the extended paper [3]. Here, we state the major theorems and lemmas. As expected, TWAM satisifies progress and preservation:

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

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

Here m fails means that a query failed in the sense that all proof rules have been exhausted—it does not mean the program has become stuck. m done means a program has terminated successfully. Soundness (Theorem 1) is a corollary:

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

Soundness follows from progress and preservation, because on m' done only holds for `succeed`, and by the typing rule for `succeed`.

We overview major lemmas, including all those discussed so far:

- Static unification computes most-general unifiers.
- Language constructs obey their appropriate substitution lemmas, even in the presence of unification.
- Dynamic unification is sound with respect to static unification.
- When the occurs check passes, binding a variable does not introduce cycles.
- Updating the trail maintains trail invariants and backtracking maintains machine state invariants.

Our notion of correctness for static unification follows the standard correctness property for first-order unification: we compute the most general unifier, i.e., a substitution which unifies M_1 with M_2 and which is a prefix of all unifiers.

Lemma (Unify 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* $\sigma' = \sigma^*, \sigma$ *up to alpha-equivalence.*

While this lemma is standard, it is essential to substitution. While we have numerous substitution lemmas (e.g. for heaps), we mention the lemma for instruction sequences here because it is the most surprising.

Lemma (I-Substitution)**.** *If* $\Delta_1, x{:}A, \Delta_2; \Gamma \vdash I$ ok *and* $\Delta_1 \vdash M{:}A$ *then we have* $\Delta_1, [M/x]\Delta_2; [M/x]\Gamma \vdash [M/x]I$ ok

The most challenging cases are those involving unification. Unification is not always preserved under substitution; in this case, $[M/x]I$ is vacuously well-typed as discussed in Section 3.3. In the case where unification is preserved, we exploit the fact that the derivation for I computed the *most general* unifier, which is thus a prefix of the unifier from $[M/x]I$. At a high level, this suffices to show all necessary constraints were preserved by substitution.

The progress and preservation cases for unification instructions need to know that dynamic unification unify is in harmony with static unification.

Lemma (Soundness of unify)**.** *If* $\Delta \vdash M_1 : a$ *and* $\Delta \vdash M_2 : a$ *and* $\Delta \vdash H : \Psi$ *and* $\Delta; C; H \vdash T$ ok *and* $\Delta; \Psi \vdash \ell_1 : \mathfrak{S}(M_1 : a)$ *and* $\Delta; \Psi \vdash \ell_2 : \mathfrak{S}(M_2 : a)$ *then*

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

The heap update lemma says that when the occurs check passes, the result of binding a free variable is well-typed (with the new binding reflected by a substitution into the heap type Ψ). Because the typing invariant implies acyclic heaps, this lemma means cycles are not introduced.

Lemma (Heap Update)**.** *If* $\Delta \vdash H : \Psi$ *and* $\Psi(\ell_1) = \mathfrak{S}(x : a)$ *then*

(a) *If* $\Psi(\ell_2) = \mathfrak{S}(M : a)$ *and* $\ell_1 \notin_H \ell_2$, *(the occurs check passes) then* $\Delta \vdash H\{\ell_1 \mapsto \textbf{\textit{BOUND}}\ \ell_2\} : [M/x]\Psi$.

(b) *If for all* i, $\Psi(\ell'_i) = \mathfrak{S}(M_i : a_i)$ *and* $\ell_1 \notin_H \ell'_i$ *and* $\Sigma(c) = \boldsymbol{a} \to a$, *then* $\Delta \vdash H\{\ell_1 \mapsto c\langle \ell'_1, \ldots, \ell'_n\rangle\} : [c\ \boldsymbol{M}]\Psi$.

This lemma is more subtle than its statement suggests, and demonstrates the subtle relationship between heaps, heap types, and heap typing derivations. Recall that heaps and heap types are unordered: the typing derivation itself exhibits a topological ordering as a witness that there are no cycles. The proof of Heap Update is constructive and proceeds by induction on the derivation: an algorithm can be given which computes a new topological ordering for the resulting heap.

Introducing free variables and binding free variables both preserve the validity of the trail:

Lemma (Trail Update)**.** *If* $\Delta; C; H \vdash T$ *ok then*

(a) *If* $H(\ell^H) = \textbf{\textit{FREE}}[x : a]$ *then* $\Delta; H\{\ell^H \mapsto w\} \vdash \mathit{update\_trail}(x : a@\ell^H, T)$ *ok*.

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

Claim (a) says that if we bind a free variable x to a term and add x to the trail (notated $x : a@\ell^H$ to indicate a variable x of type a was located at ℓ^H), the resulting trail is well-typed. The trail $\mathsf{update\_trail}(x : a@\ell^H, T)$ is well-typed under the heap $H\{\ell^H \mapsto w\}$ iff unwinding the trail $\mathsf{update\_trail}(x : a@\ell^H, T)$ results in some well-typed store S'. Thus the proof of claim (a) amounts to showing that unwinding $\mathsf{update\_trail}(x : a@\ell^H, T)$ gives us the original store S, which we already know to be well-typed.

Claim (b) is a weakening principle for trails, which comes directly from the weakening principle for heaps (a heap $H : \Psi$ is allowed to contain extra unreachable locations ℓ which do not appear in Ψ). 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. It relies on the following subclaim, which holds by induction on *tf*.

Claim. $\mathsf{unwind}(\Delta, x : a, H\{\{\ell^H \mapsto \mathbf{FREE}[x : a]\}\}, \mathit{tf}) = (\Delta, H'\{\{\ell^H \mapsto \mathbf{FREE}[x : a]\}\})$ for some H'.

Recall that the typing rule for trails simply says whatever state results from unwinding must be valid. This simplifies the proofs significantly: showing that an update preserves validity consists simply of showing that it does not change the result of backtracking (modulo perhaps introducing unused values).

Soundness of the backtracking operation simply says the resulting machine is well-typed. The proof is direct from the premisses of the trail typing invariant.

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

While the full proof contains several dozen other lemmas, those discussed above constitute the major insights why the TWAM type system is sound and constitutes certification for TWAM programs.

4 Conclusion

To address the problem of constructing verified compilers for logic programs, we have designed and implemented a *certifying abstract machine* called the Typed Warren Abstract Machine, or TWAM. Our metatheory proves that type-checking for the TWAM constitutes certification of partial correctness with respect to an LF signature. We have demonstrated the viability of this approach by implementing a compiler from T-Prolog to the TWAM, which we have made available at `http://www.cs.cmu.edu/~bbohrer/pub/twam.zip`. Our implementation is approximately 5,000 lines of Standard ML, of which approximately 400 are the TWAM typechecker, which constitutes the trusted core. We have tested our implementation on a small arithmetic library (a few hundred lines). Even this small test exercises all the features of T-Prolog and helped us fix compiler bugs during development, e.g. indexing errors in our register allocator which might not have been detected by weaker type systems.

Our work differs from previous work on logic program compilation because we are the first to take a certifying compilation approach. We have also produced a working compiler with a formal guarantee, whereas previous efforts stopped before implementing a compiler [24, 4, 2, 22]. Several optimizing compilers have been verified in proof assistants [12, 11] and some of them use proof-producing compilation [17], but these do not address logic programming languages. Our work can also be seen as falling into the paradigm of type-preserving compilation [25], an instance of certifying compilation [18] where the certificates are typing information. Type-preserving compilers have similar strengths and weaknesses to other certifying compilers.

Our type system relies on the logical framework LF [10], and is inspired by other dependently- typed languages [28], 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].

This work has several avenues for future improvement. While the strength of this work is a novel and powerful certification approach supported by sound theoretical foundations, we have ignored some aspects relevant in practice, such as performance of generated code. The WAM supports a well-known set of optimizations that have a significant impact in practice [1]. Adding support for these optimizations to the TWAM would open the path to acheiving the ultimate goal: a production-quality optimizing compiler certified by the TWAM. To broaden its applicability we also wish to extend TWAM to support logic programming

languages such as Elf [19] with advanced type system features. Going even further, one could explore whether the certifying abstract machine approach can benefit compilers for languages outside the logic programming paradigm.

References

1. Aït-Kaci, H.: Warren's abstract machine: A tutorial reconstruction. MIT Press (1991)
2. Beierle, C., Börger, E.: Correctness proof for the WAM with types. Computer Science Logic, LNCS volume 626 pp. 15–34 (1992)
3. Bohrer, B., Crary, K.: TWAM: A certifying abstract machine for logic programs. CoRR **abs/1801.00471** (2018), http://arxiv.org/abs/1801.00471
4. Börger, E., Rosenzweig, D.: The WAM—definition and compiler correctness. Logic Programming: Formal Methods and Practical Applications (1994)
5. Cervesato, I.: Proof-Theoretic Foundation of Compilation in Logic Programming Languages pp. 115–129
6. Crary, K.: Toward a foundational typed assembly language. POPL '03 pp. 198–212 (2003). https://doi.org/10.1145/604131.604149
7. Crary, K., Sarkar, S.: Foundational certified code in the Twelf metalogical framework. ACM Trans. Comput. Logic **9**(3), 16:1–16:26 (Jun 2008). https://doi.org/10.1145/1352582.1352584
8. Crary, K., Vanderwaart, J.C.: An expressive, scalable type theory for certified code. ICFP pp. 191–205 (2002)
9. Elliott, C., Pfenning, F.: A semi-functional implementation of a higher-order logic programming language. In: Topics in Advanced Language Implementation. pp. 289–325. MIT Press (1991)
10. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. JACM (1993)
11. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: A verified implementation of ML. POPL '14 pp. 179–191 (2014). https://doi.org/10.1145/2535838.2535841
12. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. POPL '06 (2006)
13. Li, G., Owens, S., Slind, K.: Structure of a proof-producing compiler for a subset of higher order logic. ESOP (2007)
14. Martelli, A., Montanari, U.: An efficient unification algorithm. TPLS **'82**(2), 258–282 (Apr 1982). https://doi.org/10.1145/357162.357169
15. Morrisett, G., Crary, K., Glew, N., Walker, D.: Stack-based typed assembly language. JFP **'02**, 43–88 (Jan 2002). https://doi.org/10.1017/S0956796801004178
16. Morrisett, G., Walker, D., Crary, K., Glew, N.: From System F to typed assembly language. TPLS **21**(3), 527–568 (May 1999). https://doi.org/10.1145/319301.319345
17. Myreen, M.O., Owens, S.: Proof-producing translation of higher-order logic into pure and stateful ML. JFP **'14**, 284–315 (May 2014)
18. Necula, G.C., Lee, P.: The design and implementation of a certifying compiler. ACM SIGPLAN Notices **33**(5), 333–344 (1998)
19. Pfenning, F.: Elf: A language for logic definition and verified metaprogramming. LICS pp. 313–322 (1989)
20. Pfenning, F.: Unification and anti-unification in the calculus of constructions. LICS pp. 74–85 (1991)

21. Pfenning, F., Schürmann, C.: System description: Twelf–a meta-logical framework for deductive systems. CADE **'99**, 202–206 (1999)
22. Pusch, C.: Verification of compiler correctness for the WAM. TPHOLs '96 pp. 347–362 (1996)
23. Robinson, J.A.: A machine-oriented logic based on the resolution principle. JACM **'65**(1) (1965). https://doi.org/10.1145/321250.321253
24. Russinoff, D.M.: A verified Prolog compiler for the Warren abstract machine. Journal of Logic Programming **'13**, 367–412 (1992)
25. Tarditi, D., Morrisett, J.G., Cheng, P., Stone, C.A., Harper, R., Lee, P.: TIL: A type-directed optimizing compiler for ML. In: PLDI. pp. 181–192. ACM (1996)
26. Warren, D.H.: An abstract Prolog instruction set, vol. 309. Artificial Intelligence Center, SRI International Menlo Park, California (1983)
27. Wielemaker, J.: SWI-Prolog OpenHub project page. https://www.openhub.net/p/swi-prolog (2018), accessed: 2018-04-28
28. Xi, H., Pfenning, F.: Dependent types in practical programming. POPL '98 pp. 214–227 (1998)
29. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. SIGPLAN Not. **46**(6), 283–294 (Jun 2011). https://doi.org/10.1145/1993316.1993532