TWAM: A Certifying Abstract Machine for Logic Programs

Brandon Bohrer(✉) and Karl Crary

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 typed compiler for an idealized 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 show programs obey a semantics based on provability in first-order logic (FOL). We present a soundness metatheorem which (going beyond the guarantees provided by most typed compilers) constitutes a partial behavior correctness guarantee: well-typed TWAM programs are sound proof search procedures with respect to a FOL signature. We argue why this guarantee is a natural choice for significant classes of logic programs. 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 introduce, 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. The most successful compilers [16,17] use an approach we will call *direct verification*, showing that compilation of any valid program results in a refinement thereof. Multiple approaches have been tried for logic programming, but none have resulted in a executable verified compiler for logic programs.

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. Beyond the intellectual interest in compiler correctness, there is a practical concern for correctness of logic program compilation: practical implementations can be large. For example, SWI-Prolog is estimated at over 600,000 lines of code [36]. While our certifying compiler is much smaller, it provides a natural first step toward production-scale verification.

R. Piskac and P. Rümmer (Eds.): VSTTE 2018, LNCS 11294, pp. 112–134, 2018.
https://doi.org/10.1007/978-3-030-03592-1_7

Certifying compilation [23] is an approach to verification wherein the compiler outputs a formal certificate (in our case, type annotations) that the compiled program satisfies some desired property. Certifying compilation, unlike direct verification, has the advantage that the certificates can be distributed with the compiled code and checked independently by third parties, which is useful, e.g., for ascertaining trust in code downloaded from the Web. Additional engineering advantages include the ability to write multiple independent checkers for improved confidence, to share a certificate language between multiple compilers for the same language, or even to share the certificate language between compilers for different languages so long as the target language and specification language are suitable for both (e.g. they have similar dynamic semantics but different static semantics). 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.

Traditionally, the other cost of certifying compilation [23] is that only type and memory safety are certified, not dynamic correctness. In contrast, we certify *search soundness*, which is a non-trivial dynamic correctness property. This leap has only been made recently in the context imperative and functional languages [6,14]. We provide their logic programming counterpart. The sense in which we do so is made precise in Theorem 1.

Theorem 1 (*Search Soundness*): Let P be a logic program and Q a query formula. If query `?-` Q. succeeds on program P, then $P \vdash \sigma(Q)$ is derivable in first-order (minimal) logic for some substitution σ.

We choose not to certify completeness with respect to, e.g., Prolog's depth-first semantics. For important classes of programs (typecheckers, proofcheckers, expert systems), soundness is fundamental: Checkers should accept only valid programs and valid proofs, while expert systems should provide only justified advice. Otherwise, a user might run an unsafe program, believe an untrue statement, or take unreasonable actions. Theorem 1 says that all such guarantees which hold of the source transfer to the compiled code. While completeness is desirable, soundness is our priority because preventing undesired behavior is often more impactful than ensuring desired behavior.

Ignoring completeness is valuable because it allows us to use provability as the semantics of logic programs, abstracting over operational details like proof search order. This imprecision is sometimes a feature, e.g. when we wish to let the compiler reorder clauses for performance.

In Theorem 1, the logic programs are programs in our T-Prolog language. In order to keep the correspondence with first-order logic close, T-Prolog enforces the occurs check and removes cut and negation-as-failure[1]. T-Prolog also supports (simple) inductive data types. Since untyped Prolog is most familiar, our examples are untyped Prolog, or equivalently all terms have the same type `term`.

The heart of this work is the development of our compilation target, the Typed Warren Abstract Machine (TWAM), a dependently-typed *certifying*

[1] See Sect. 6 for how these features might be supported.

abstract machine for logic programs, inspired by the Warren Abstract Machine (WAM) [35]. TWAM diverges from WAM in several ways to simplify our formal development: (1) we use continuation-passing style (CPS) 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 Sect. 3, soundness of the TWAM type system says that well-typed programs are sound proof search procedures for their first-order logic (FOL) signature. We have implemented a compiler from T-Prolog to TWAM and an interpreter for TWAM code, which we have tested on a small library of 468 lines. The result is a certifying compiler with a special-purpose proof checker as its trusted core: the TWAM typechecker.

Background: Dependent Types and Proof Terms. Our type system integrates first-order (minimal) logic (FOL) to specify the semantics of logic programs. We use the variable M to range over FOL terms, D to range over FOL proofs, a to range over (simple) types, and A to range over propositions. The type-theoretic analog of the quantifier $\forall x{:}a.\ \phi$ is the dependent function type $\Pi x{:}a.\tau$. When the name x is not referenced in τ, this is equivalent to the simple function type $a \Rightarrow \tau$. We borrow some notations from the logical framework LF [13], a type system corresponding to first-order minimal logic. Our proof language is minimalistic, consisting of constants, modus ponens, instantiation, abstraction, and variables. More information about proof terms for first-order logic is available in Sørensen [32, Chap. 8]. We refer to proof terms D as just "proofs" to avoid confusion with simply-typed FOL terms M. We use juxtaposition $D\ D$ to indicate modus ponens and $D\ M$ for universal quantifier instantiation. Abstraction $\lambda x.\ D$ can range over either proofs $x{:}A$ or term $x{:}a$. We write $\lambda x{:}\alpha.\ D$ wherever both abstraction over proofs and abstraction over individuals are permissible. Similarly, we write θ where both terms M and proofs D may appear.

$$\begin{array}{lll} \text{FOL propositions} & \text{A} ::= c & \mid \forall x : a.\ A \mid A \Rightarrow A \mid A\ M \\ \text{FOL terms} & \text{M} ::= x & \mid c \mid M\ M \\ \text{FOL proofs} & \text{D} ::= x & \mid c \mid D\ \theta \mid \lambda x : \alpha.\ D \end{array}$$

In Sect. 2 we will extract a constant c for each type, constructor, and clause of a program. This collection of constant declarations is called a *FOL signature*, written Σ. The compiler generates a FOL signature from an arbitrary T-Prolog program. This signature provides a precise formal specification of what it means for proof search to find a valid proof.

While we attempt to introduce key WAM concepts as we go, unfamiliar readers will benefit from reading Aït-Kaci [1]. A gentler version of this paper with extended proofs, definitions, and a simply-typed variant of WAM is available [4].

2 Certifying Compilation in Proof-Passing Style

We briefly demonstrate (Fig. 1) the T-Prolog source syntax and the extraction of a FOL signature Σ from a T-Prolog program. We consider addition on the Peano naturals as a running example, i.e., a predicate `plus(N_1,N_2,N_3)` that holds when $N_1 + N_2 = N_3$. We write 0 and 1+ for the Peano natural constructors. We

also write, e.g., 1 as shorthand for 1+ 0. A T-Prolog program consists of standard Prolog syntax plus optional type annotations. Throughout the paper, we write vectors in bold, e.g.; $\boldsymbol{a}$ below. Throughout, a ranges over simple (inductive) types while A is ranges over propositions in FOL. All terms in the untyped fragment of T-Prolog have a distinguished simple type, `term`.

- A type a in T-Prolog translates to a FOL type a.
- A term constructor $c\colon \boldsymbol{a} \to a$ translates to a FOL term constructor of the same type. For untyped c, the result and all arguments translate to `term`.
- A predicate $p\colon \boldsymbol{a} \to \texttt{type}$ translates to a FOL constant $P\colon \boldsymbol{a} \to \mathbb{B}$ where $\mathbb{B}$ is the type of booleans.
- A clause C of form G `:-` $\mathrm{SG}_1, \ldots, \mathrm{SG}_n$. translates to a FOL proof constructor $c\colon \forall_{\mathbf{FV}(C)}.\ \boldsymbol{SG} \to G$ where $\mathbf{FV}(C)$ is the set of free variables of clause C and $\boldsymbol{SG}$ consists of one argument for each subgoal. This is the universal closure of the Horn clause $\mathbf{SG} \to G$.
- The query `?-` Q is translated to a distinguished predicate named *Query* with one proof constructor $\mathit{QueryI} : \forall_{\mathbf{FV}(Q)}.\ Q \to \mathit{Query}$[2].

	$0\colon \texttt{term}$
(no type declarations)	$1+ : \texttt{term} \Rightarrow \texttt{term}$
	$\mathit{Plus}\colon \texttt{term} \Rightarrow \texttt{term} \Rightarrow \texttt{term} \Rightarrow \mathbb{B}$
`plus(0,N,N).`	$\mathit{Plus\text{-}Z}\colon (\forall N\colon \texttt{term}.\ \mathit{Plus}\ 0\ N\ N)$
`plus(1+(N_1),N_2,1+(N_3)) :-`	$\mathit{Plus\text{-}S}\colon \big(\forall N_1\colon \texttt{term}.\ \forall N_2\colon \texttt{term}.\ \forall N_3\colon \texttt{term}.$
`  plus(N_1,N_2,N_3).`	$\quad \mathit{Plus}\ N_1\ N_2\ N_3 \to$
	$\quad\quad \mathit{Plus}\ (1+\ N_1)\ N_2\ (1+\ N_3)\big)$
`?- plus(N_1, 0, 1+(0)).`	$\mathit{QueryI}\ :(\forall N_1\colon \texttt{term}.\ \mathit{Plus}\ N_1\ 0\ 1 \to \mathit{Query})$

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

The TWAM certification approach can be summed up in a slogan:

Typed Compilation + Programming As Proof Search = Proof-Passing Style

Typed compilation uses the type system of the target language to ensure that the program satisfies some property. Previous work [34] has used typed compilation to ensure intermediate languages are safe (do not segfault). One of our insights is that by combining this technique with the programming-as-proof-search paradigm that underlies logic programming, our compiler can certify a much stronger property: search soundness (Theorem 1).

A TWAM program must contain enough information that the TWAM type-checker can ensure that a proof of the query exists for each terminating runs of the program. We achieve this by statically ensuring that whenever *each* proof

[2] Note that *Query* has no free variables. This simplifies the proof of Theorem 1 because it depends heavily on substitution reasoning.

search procedure p returns, the corresponding predicate P will have a proof in FOL. This amounts to (1) annotating each return point with the corresponding FOL proof and (2) reasoning statically about constraints on T-Prolog terms with dependent *singleton types* $\mathfrak{S}(M : a)$ containing exactly the values that represent some FOL term M of simple type a[3]. Singleton typing information is needed to typecheck almost any FOL proof term. For example, an application of *Plus-Z* only checks whether we statically know that 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 be inspected at runtime and thus can be (and in our implementation, are) *erased* before execution. In the following syntax, we annotate all erasable type annotations and subterms with square brackets. The Simply-Typed WAM [4] shows how TWAM works after erasure. Because proofs are only performed during typechecking, they have no (direct) runtime overhead, compared to runtime proof computations, which are expensive. At the same time, we do simplify WAM (e.g. with heap-allocated environments) in order to make developing a type system more feasible. For this reason, we do not expect our current implementation to be competitive with production compilers.

3 The Typed WAM (TWAM)

In this section, we develop the main theoretical contributions of the paper: the design and metatheory of the TWAM. We begin by introducing the syntax and operational semantics of TWAM by example. We then develop a type system for TWAM which realizes proof-passing style. We give an outline of the metatheory, culminating in a proof (Sect. 3.5) of Theorem 1.

3.1 Syntax

We begin by presenting the syntaxes for TWAM program texts, machine states (as used in the operational semantics), and typing constructs, which are given in Fig. 2. We call the formal representation of a TWAM program text a *code section* C. Each basic block in the program has its own identifier ℓ^C; the code section maps identifiers to *code values*, which we range over with variable v^C. Code values are always of the form $[\lambda \boldsymbol{x} : \boldsymbol{\alpha}.]\texttt{code}[\Gamma](I)$ where I is a basic block (instruction sequence), $\lambda \boldsymbol{x} : \boldsymbol{\alpha}$ (possibly empty) specifies any FOL (term and/or proof) parameters of the basic block, and Γ is a *register file type* specifying the expected register types at the beginning of the basic block. Recall that the square brackets above indicate that λ-abstractions and type annotations are needed only for certification and that because they do not influence the operational semantics,

[3] Our running example is untyped ($a = \texttt{term}$ throughout) because untyped Prolog is well-known, but we will still present the typing rules in their full generality.

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

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

they can be type-erased before execution. Note that when the λ-abstractions are type erased, their matching (FOL) function applications will be as well. Brackets also appear in the syntax of machine states (e.g., $\mathbf{FREE}[x : a]$): these too are erased because they are used only in the metatheory and are not required at runtime. Recall also that Π is a dependent function type, which is analogous to the quantifier $\forall x.\ \phi$.

3.2 Example: Code Section for `Plus`

We continue the running example: we present a code section which contains the implementation of the `plus` proof search procedure, consisting of two code values named `plus-zero/3` and `plus-succ/3`. Like all TWAM code, it is written in continuation-passing style (CPS): code values never return, but rather return control to the caller by invoking a success continuation passed in to the callee through a register. The code section also includes an implementation of an example query, `plus(N, 0, 1+(0))`, consisting of a code value named `query/0`. When the query succeeds, it invokes the top-level success continuation, which is a code value named `init-cont/0`.

As is typical in continuation-passing-style, code values have no return type because they never return. The type of a code value is written $\Pi \boldsymbol{x}{:}\boldsymbol{\alpha}.\ \neg\Gamma$, where $\boldsymbol{x}{:}\boldsymbol{\alpha}$ records any FOL terms and proofs passed as static arguments, while Γ records any heap values passed at runtime through the register file.

Example 1 (Implementing `plus`)

```
# Entry point to plus, implements
# case plus(0,N,N) and tries
# plus-succ/3 on failure
plus-zero/3 ↦ [λN1N2N3: term.]code[
  {A1 : S(N1), A2 : S(N2), A3 : S(N3),
    ret: ((Plus N1 N2 N3) ⇒ ¬{})}](
  put_tuple X1, 4;
    set_val A1;
    set_val A2;
    set_val A3;
    set_val ret;
  push_bt X1, (plus-succ/3 [N1 N2 N3]);
  get_str A1, 0;
  get_val A2, A3;
  jmp (ret [(Plus − Z N2)]))
```

```
# plus(1+(N_1), N_2, 1+(N_3))
#    :- plus(N_1,N_2,N_3).
plus-succ/3 ↦ [λN1N2N3: term.]code[
  {env: x[S(N1), S(N2), S(N3),
    ((Plus N1 N2 N3) ⇒ ¬{})]}](
  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 N_1-1
    unify_var A1, [NN1 : term.]
  get_str A3, 1+;
  #Set arg 3 of rec. call to N_3-1
    unify_var A3, [NN3 : term.]
  #tail-call optimization: add
  #Plus-S constructor when called
  mov ret, [(λD: Plus NN1 N2 NN3.]
    ret [(Plus-S NN1 N2 NN3 D)]);
  jmp (plus-zero/3 [NN1 N2 NN3]))
```

Example 2 (Calling `plus`)

```
init-cont/0 ↦
  [λN: term. λD: (Plus N 0 (1+ 0)).]
  code[{}](succeed[(QueryI N D):Query])
```

```
# plus(N, 0, 1+(0))
query/0 ↦ code[{}](
  put_var A1, [N: term].
  put_tuple X1, 0;
  close ret, X1, (init-cont/0 [N]);
  put_str A2, 0;
  put_str A3, 1+;
    unify_val A2, [_: term.]
  jmp (plus-zero/3 [N 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 $A_1 = 0$. Its type annotation states that the argument terms N_1 through N_3 are 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* N_1 N_2 N_3 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 location stored in `ret`, proving *Plus* N_1 N_2 N_3 in FOL 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 $NN_1 + N_2 = NN_3$, where NN_1 and NN_3 are the predecessors of N_1 and N_3. 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* NN_1 N_2 NN_3 implies *Plus* N_1 N_2 N_3, deriving the latter also requires applying *Plus-S* after

proving the former. This `mov` instruction simply says to apply *Plus-S* (statically) before invoking `ret`. Because only the proof changes, the `mov` can be erased before executing the program.

Machines. As shown in Fig. 2, the state of a TWAM program is formalized as a tuple $m = (\Delta, T, C, H, R, I)$ (or a special machine `read` or `write`: see, e.g., Sect. 3.3). 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 (location 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 *heap* H and *code section* C have types notated Ψ and Ξ, $R : \Gamma$ is the register file, and I represents the program counter as the list of instructions left in the current basic block. Typical register names are A_i for arguments, X_i for temporaries, `ret` for success continuations, and `env` for environments. Δ contains the free term variables of H; it is used primarily in Sect. 3.5. 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 FOL) c applied to arguments $\langle \ell_1, \ldots, \ell_n \rangle$. As in WAM, the heap is in disjoint-set style, i.e. all free variables are distinct and pointers $\mathbf{BOUND}\ \ell$ can be introduced when unifying variables; $\mathbf{BOUND}\ \ell$ and ℓ represent the same FOL 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.

3.3 Operational Semantics

We give the operational semantics by example. Due to space constraints, see the extended paper [4] 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(N,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. If we wish, we can view the final instruction of a write-mode spine as two evaluation steps (delimited by a semicolon), one of which constructs the last argument of the constructor and one of which combines the arguments into a structure. 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 to be acyclic when the occurs check passes (should the occurs check fail, we backtrack instead). Below, all occurs checks pass, and are omitted for brevity. Spines, backtracking, and no-ops are marked in monospace.

```
   query/0 ↦ code[{}](                                  Outcome:
 1 put_var A1,[N:term].                                 H←H{{ℓ1↦FREE[N:term]}}, R←R{A1↦ℓ1}
 2 put_tuple X1,0;                                      H←H{{ℓ2↦()}}, R←R{X1↦ℓ2};
 3 close ret,X1,(init-cont/0[N]);                       H←H{{ℓ3↦close(init-cont/0 [N],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[_: term]}},
                                                        R←R{A3↦ℓ5}, c = 1+; ℓ = ℓ5, ℓs = ⟨⟩
 6   unify_val A2,[_: term.]                            ℓs←⟨ℓ4⟩; H←H{ℓ5↦1 + ⟨ℓ4⟩}
 7 jmp plus-zero/3[···]                                 I←(C(plus-zero/3) [N 0 (1+ 0)])

   plus-zero/3: ([λN1N2N3 : term.]
   code[{A1 : S(N1), A2 : S(N2), A3 : S(N3), ret : ((Plus N1 N2 N3) ⇒ ¬{})}](
 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[···]);                       T←(plus-succ/3[N1 N2 N3], ℓ6, ⟨⟩) :: ⟨⟩
14 get_str A1,0;                                        WRITE:H←H{ℓ1↦0},
                                                        T←(plus-succ/3[N1N2N3], ℓ6, ⟨ℓ1⟩) :: ⟨⟩
15 get_val A2,A3;                                       BT:T←⟨⟩, I←plus-succ/3 . . . ,
                                                        H←H{ℓ1↦FREE[N : term]}

   plus-succ/3 ↦ [λN1N2N3 : term.]
   code[{env : x[S(N1), S(N2), S(N3), (Plus N1 N2 N3) ⇒ ¬{}]](
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,[NN1 : term.]                         H←H{{ℓ7↦FREE[NN1 : term]}}
                                                        R←R{A1↦ℓ7}, ℓs = ⟨ℓ7⟩;
                                                        H←H{ℓ1↦1+ ⟨ℓ7⟩}
22 get_str A3,1+;                                       READ:ℓs = ⟨ℓ4⟩
23   unify_var A3,[NN3 : term.]                         R←R{A3↦ℓ4}
24 mov ret,[(λD : (Plus NN1 N2 NN3).]ret[(Plus − S NN1 N2 NN3 D)]);
                                                        NOP:R←R{{ret↦[(λD:(Plus NN1 N2 NN3).]
                                                          ℓ3[(Plus − S NN1 N2 NN3 D)])}}
25 jmp (plus-zero/3 [···]));                            I←C(plus-zero/3) [NN1 N2 NN3]

   plus-zero/3 ↦ [λN1N2N3 : term.]
     code[{A1 : S(N1), A2 : S(N2), A3 : S(N3), ret : ((Plus N1 N2 N3) ⇒ ¬{})}](
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, λ . . . ℓ3⟩;
                                                        H←H{{ℓ8↦(ℓ7, ℓ4, ℓ4, λ . . . ℓ3)}}
                                                        R←R{X1↦ℓ8}
31 push_bt X1,(plus-succ/3 [···]);                      T←(plus-succ/3 [N1 N2 N3], ℓ8, ⟨⟩) :: ⟨⟩
32 get_str A1,0;                                        READ: ℓs = ⟨⟩, ℓ = ℓ7; H←H{ℓ7↦0}
33 get_val A2,A3;                                       NOP:R(A2) = R(A3)
34 jmp (ret(Plus − Z N2));                              I←C(R(ret)) (Plus − Z N2)
                                                        = C(init-cont/0) [0 N2 N3
                                                        (Plus − S 0 N2 NN3 (Plus − Z N2))])

35 init-cont/0:[λN:term D:(Plus N 0 1).]code[{}](succeed[(QueryI N D):Query])
```

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 that if proof search succeeds, we will return to `init-cont/0`, which immediately ends the program in success. Line 4 allocates the number 0 at ℓ_4. Lines 5–6 are a write spine that constructs $1+0$. Because A_2 already contains 0, we can eliminate a common subexpression, reusing it for $1+0$. This is an example of an optimization that is possible in the TWAM. 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 on, the trace will be updated any time we bind a free variable. Line 14 dynamically checks A_1, observes that 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 trailed locations to free variables, and loading an environment. The `plus-succ/3` case proceeds successfully: 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.

3.4 Statics

This section presents the TWAM type system. The main typing judgement $\Delta; \Gamma \vdash I_{\Sigma;\Xi}$ ok says that instruction sequence I is well-typed. We omit the signature Σ and code section type Ξ when they are not used. A code section is well-typed if every block is well-typed. The system contains a number of auxiliary judgments, which will be introduced as needed. Note that the judgement $\Delta; \Gamma \vdash I_{\Sigma;\Xi}$ ok is not parameterized by the query directly; instead, the query is stored as $\Sigma(\mathit{Query})$. The typing rule for `succeed` then looks up the query in Σ to confirm that proof search proved the correct proposition. Below, the notation $\Psi\{\ell\colon \tau\}$ denotes the heap type Ψ with the type of ℓ replaced by τ whereas $\Psi\{\{\ell\colon \tau\}\}$ denotes Ψ extended with a fresh location ℓ of type τ.

Success. We wish to prove that a program only succeeds if a proof D of the *Query* exists in FOL. We require exactly that in the typing rule:

$$\frac{\Delta \vdash D\colon \mathit{Query}}{\Delta; \Gamma \vdash \texttt{succeed}[D\colon \mathit{Query}]; I\ \texttt{ok}}\ \textsc{Succeed}$$

The `succeed` rule is simple, but deceptively so: the challenge of certifying compilation for TWAM is how to satisfy the premiss of this rule. The proof-passing approach says we satisfy this premiss by threading FOL proofs statically through every predicate: by the time we reach the `succeed` instruction, the proof of the query 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 (statically) passes a FOL proof to the success continuation. These FOL proofs are part of the `jmp` instruction's *operand* op:

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

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

The *operands* consist of locations, registers, FOL applications, and FOL abstractions:

operands op $::= \ell \mid r \mid op\ [\theta] \mid [\lambda x\colon \alpha.]\ op$

Operand typechecking is written $\Delta; \Gamma \vdash op\colon \tau$ and employs standard rules for checking FOL terms. Brackets indicate that argument-passing and λ-abstraction are type-erased. 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` $r_d, r_e, \ell^C[\boldsymbol{\theta}]$ constructs a closure in r_d which, when invoked, executes the instructions at ℓ^C using FOL arguments $\boldsymbol{\theta}$ and environment r_e. The environment is an arbitrary value which is passed to $\ell^C\ [\boldsymbol{\theta}]$ in the register `env`. The argument $(\ell^C\ [\boldsymbol{\theta}])$ is an operand, syntactically restricted to be a location applied to arguments.

$$\frac{\Gamma(r_e) = \tau \quad \Delta; \Gamma\{r_d\colon \Pi \boldsymbol{x}\colon \boldsymbol{\alpha}.\ \neg\Gamma'\} \vdash I\ \texttt{ok}}{\Delta; \Gamma \vdash \texttt{close}\ r_d, r_e, (\ell^C\ [\boldsymbol{\theta}]); I\ \texttt{ok}}\ \textsc{Close}$$

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

$$\frac{\Delta; \Gamma \vdash I\ \texttt{ok} \quad \Gamma(r_e) = \tau \quad \Delta; \Gamma \vdash (\ell^C\ [\boldsymbol{\theta}])\colon \neg\{\texttt{env}\colon \tau\}}{\Delta; \Gamma \vdash \texttt{push\_bt}\ r_e, (\ell^C\ [\boldsymbol{\theta}]); I\ \texttt{ok}}\ \text{BT}$$

Singleton Types. The Putvar rule introduces a FOL variable x of simple type a, corresponding to a T-Prolog unification variable. Statically, the FOL 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.

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

Singleton typing knowledge is then exploited in proof-checking FOL proofs.

Unification. However, `put_var` alone does not provide nearly enough constraints to check most proofs. Almost every FOL proof 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 FOL proofs. We separate unification into a judgement $\Delta \vdash M_1 \sqcap M_2 = \sigma$ which computes a 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 variable substituted by 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 [18,29]. One such algorithm is given in the extended paper [4].

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. This is one of the major 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 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.

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

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

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

$$\frac{\Gamma(r_s) = \mathrm{x}[\boldsymbol{\tau}]\Gamma\{r_d\colon \tau_i\} \vdash I\ \mathtt{ok} \quad (\text{where } 1 \le i \le |\boldsymbol{\tau}|)}{\Delta; \Gamma \vdash \mathtt{proj}\ r_d, r_s, i; I\ \mathtt{ok}}\ \textsc{Proj}$$

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 $\Delta; \Gamma \vdash_{\Sigma;\Xi} I{:}J_t$ where J_t is a *tuple spine* type with form

$\boldsymbol{\tau_2} \Rightarrow \{r_d{:}\mathrm{x}[\boldsymbol{\tau_1}\boldsymbol{\tau_2}]\}$. 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 $\Delta; \Gamma \vdash I$ ok when the spine completes.

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

$$\frac{\Gamma(r) = \tau \Gamma \vdash I{:}J_t}{(\Delta; \Gamma \vdash \texttt{set\_val}\ r; I) : (\tau \to J_t)}\ \textsc{TSpine-SetVal}$$

$$\frac{\Delta; \Gamma\{r_d{:}\,\tau\} \vdash I\ \mathsf{ok}}{\Delta; \Gamma \vdash I : \{r_d : \tau\}}\ \textsc{TSpine-End}$$

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 $\Gamma \vdash I_{\Sigma;\Xi} : J$ 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 FOL 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.

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

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

The instruction `get_str` unifies its argument with a term $c\ M_1\ \cdots\ M_n$ by executing a spine as described above. The `put_str` instruction starts a spine that (always) constructs a new structure.

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

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

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 that machines include a context Δ containing the free variables of the heap H. We can[4] identify variables of Δ with heap locations, trivially ensuring that each variable appears exactly once in the heap. 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 that 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:

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

Thus, the derivation exhibits a topological ordering of the heap, proving that it is acyclic. Section 3.5 shows this invariant is maintained because we only bind variables when the occurs check passes. The code section has no ordering constraint, in order to support mutual recursion.

Heap values for T-Prolog terms have singleton types:

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

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

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 the result of *unwinding* the trace tr (i.e. making the traced variables free again), is well-typed.

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

This completes the invariants for non-spinal machines.

Each of the typing invariant rules for spinal machines has an additional premiss, either $\Delta; \Psi \vdash \boldsymbol{\ell} \ \ \mathtt{reads} \ \ \Pi \boldsymbol{x} : \boldsymbol{a}.(c\ \boldsymbol{M}\ \boldsymbol{M'} \sqcap c\ \boldsymbol{M}\ \boldsymbol{x})$ (for a read spine) 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 a write spine). These are

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

some of the most complex rules in the TWAM. Nonetheless, their purpose can be explained naturally at a high level. 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 the typing judgement $\Delta; \Gamma \vdash I{:}J$.

$$\frac{\Delta;\Psi \vdash \boldsymbol{\ell} : \mathfrak{S}(\boldsymbol{M'} : \boldsymbol{a})}{\begin{array}{c}\Delta;\Psi \vdash \boldsymbol{\ell}\ \texttt{reads}\ \Pi \boldsymbol{x} : \boldsymbol{a}.\ (c\ \boldsymbol{M}\ \boldsymbol{M'} \sqcap c\ \boldsymbol{M}\ \boldsymbol{x}) \\ \Delta \vdash C : \Xi \quad \Delta \vdash H : \Psi \quad \Delta;\Gamma \vdash I : J \\ \Delta;\Psi \vdash \boldsymbol{\ell}\ \texttt{reads}\ J \quad \Delta \vdash T\ \mathsf{ok} \quad \Delta;\Psi \vdash R : \Gamma\end{array}}{\cdot \vdash_{\Sigma;\Xi} \texttt{read}(\Delta, T, C, H, R, I, \boldsymbol{\ell})\ \mathsf{ok}}$$

$$\frac{\Psi(\ell^H) = \mathfrak{S}(x' : a) \quad \Sigma(c) = \boldsymbol{a}_1 \to \boldsymbol{a}_2 \to a \quad \Delta;\Psi \vdash \boldsymbol{\ell}^H : \mathfrak{S}(\boldsymbol{M} : \boldsymbol{a}_1)}{\begin{array}{c}\Delta;\Psi \vdash (\boldsymbol{\ell}^H, \ell^H, c)\ \texttt{writes}\ \Pi \boldsymbol{x} : \boldsymbol{a}_2.\ x' \sqcap c\ \boldsymbol{M}\ \boldsymbol{x} \\ \Delta \vdash C : \Xi \quad \Delta \vdash H : \Psi \quad \Delta;\Gamma \vdash I : J \\ \Delta;\Psi \vdash (\boldsymbol{\ell}^H, \ell, c)\ \texttt{writes}\ J \quad \Delta \vdash T\ \mathsf{ok} \quad \Delta;\Psi \vdash R : \Gamma\end{array}}{\cdot \vdash_{\Sigma;\Xi} \texttt{write}(\Delta, T, C, H, R, I, c, \ell^H, \boldsymbol{\ell})\ \mathsf{ok}}$$

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

3.5 Metatheory

Proofs of metatheorems are in the extended paper [4]. Here, we state the major theorems and lemmas. As expected, TWAM satisfies progress and preservation:

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

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

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

Theorem 1 (Search Soundness)**.** *If $\cdot \vdash_{\Sigma;\Xi} m\ \mathsf{ok}$ and $m \longmapsto^* m'$ and $m'\ \mathsf{done}$ then there exists a context of term variables Δ and substitution σ such that $\Delta \vdash \sigma(Q)$ in FOL where $\Sigma(QueryI) = \forall_{\boldsymbol{FV}(Q)}(Q \to Query)$.*

Proof (Sketch). By progress and preservation, $m'\ \mathsf{ok}$. By inversion on $m\ \mathsf{done}$, have $\Delta \vdash Query$ for $\Delta = \mathbf{FV}(H)$ where H is the heap from m'. By inversion on $QueryI$, have some σ such that $\Delta \vdash \sigma(Q)$. □

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 surprisingly subtle.

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

The most challenging cases are those involving unification. Unification is not always preserved under substitution; in this case, $[\theta/x]I$ is vacuously well-typed as discussed in Sect. 3.4. 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 $[\theta/x]I$. 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 have* $\mathsf{unify}(\Delta, H, T, \ell_1, \ell_2) = \bot$
- *If* $\Delta \vdash M_1 \sqcap M_2 = \sigma$ *then have* $\mathsf{unify}(\Delta, H, T, \ell_1, \ell_2) = (\Delta', H', T')$ *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* $\Delta \vdash H\{\ell_1 \mapsto \boldsymbol{BOUND}\ \ell_2\} : [M/x]\Psi$*.*

(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*

$$(a) If\ H(\ell^H) = \boldsymbol{FREE}[x:a]\ then$$
$$\Delta; H\{\ell^H \mapsto w\} \vdash \mathsf{udate\_trail}(x : a@\ell^H, T)\ \mathsf{ok}.$$
$$(b) If\ \ell^H\ \mathsf{fresh}\ and\ x\ \mathsf{fresh}\ then\ \Delta; H\{\{\ell^H \mapsto \boldsymbol{FREE}\}\}[x:a] \vdash T\ \mathsf{ok}.$$

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 it results in a well-typed heap. Thus proving (a) amounts to showing that unwinding $\mathsf{update\_trail}(x : a@\ell^H, T)$ gives us the original heap, 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 the trace *tr* contained in *tf*.

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

Recall that the typing rule for trails simply says whatever heap results from unwinding must be well-typed. 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 demonstrate the major insights into why the TWAM type system is sound and why it enables certification for TWAM programs.

4 Implementation

Implementing a compiler from T-Prolog to TWAM, a TWAM runtime, and a TWAM typechecker allows us not only to execute T-Prolog programs, but crucially to validate the TWAM design. For example, implementation increased our confidence that the static and dynamic semantics are exhaustive. Testing the compiler and checker provides informal evidence that they are sufficiently complete in practice. Testing the checker also tests its soundness, validating simultaneously that it is faithful to the dynamics and that Theorem 1 holds of the implementation.

The proof-of-concept implementation, which consists of 5,000 lines of Standard ML, is at http://www.cs.cmu.edu/~bbohrer/pub/twam.zip. The TWAM typechecker, which constitutes the trusted core, is about 400 lines. The large majority of the core is implemented by straightforward (manual) translation of the TWAM typing rules into ML code. This is a small fraction of the code (less than 10%) and compares favorably with the trusted cores of general-purpose proof checkers. Our test suite has 23 test files totaling 468 lines, the largest of which is a library for unary and decimal arithmetic. Other files stress-test edge cases of T-Prolog and TWAM execution.

The tests showed that the TWAM checker often catches compiler bugs in practice. Many of these bugs centered around placing a value into the wrong register or wrong position of a tuple. Singleton types are effective at catching these bugs because distinct terms always have different singleton types. Prior typed intermediate languages are less certain to catch these bugs because they permit distinct terms to have the same type.

Not only did our implementation greatly increase confidence in the theory, but we believe that it demonstrates TWAM's potential for catching real bugs.

5 Related Work

We are the first to build a full certifying (or verified, in general) compiler for a Prolog-like language. In contrast, full compilers for imperative (C [17]) and functional (ML [16]) languages have been verified directly in proof assistants. The latter project also yielded a compiler from higher-order logic [21] to ML.

Compiler verification for Prolog has been explored, but past attempts did not yield a full compiler. Paper proofs were written for both concrete [30] and abstract [3,5] compiler algorithms. Some (but not all) passes of Prolog compilers were verified in Isabelle [28] and KIV [31]. Prolog source semantics have also been formalized, e.g., in Coq [15]. Compiling all the way from Prolog to WAM with proof has been noted explicitly [31] as a challenge. Previous formalized proofs reported 6 person-month development times, the same time that it took to develop our theory, proofs, and implementation. While the comparison is not direct because many details of the projects differ, we find it promising.

Certifying compilation includes type-preserving compilation [34] and proof-carrying code (PCC) [23]. In type-preserving compilation, the certificates are

type annotations, while in PCC they are proofs in logic. Type-preserving compilation is typically more concise while PCC is typically more flexible. Certifying compilation has recently been applied to the Calculus of Constructions [6] and LLVM passes [14]. A significant fragment of the proof checker for LLVM is verified in Coq for reliability. Applying this approach to TWAM is non-trivial, but possible in theory. Their experience supporting optimizations suggest we could do the same for TWAM, with proportional verification effort.

Translation validation [27] is a related approach, with post-hoc, black-box (but still automatic) construction of certificates. Its black-box nature means it might support multiple compilers, but is also often brittle.

The first-order logic we used can be embedded in the logical framework LF [13]; We have chosen FOL over LF for the simple reason that it is much better known. LF is also the foundation of the programming language Elf [25] and proof checker Twelf [26]. A comparison of our approach with Elf is fruitful: Elf instruments execution to produce LF proofs, whereas we instrument *compilation* to produce a proof that *obviates the need* for execution to produce proofs, which is amenable to higher performance. Singleton types, which are featured prominently in TWAM, are not new [38], but we are the first to support unification on singletons.

TWAM is also a descendant of typed assembly language (TAL) [7–9,19,20]. Dependent types and TAL have been combined in DTAL [37], but DTAL employs a lightweight, restrictive class of dependent types in order to, e.g., eliminate array bounds checks when compiling DML [38]. Our class of dependent types is more expressive. DTAL typechecking also requires complex non-syntactic constraint generation and solving. While TWAM's unification constraints are non-trivial, they are syntactic and thus more likely to scale.

Abstraction interpretation for Prolog [33] provides another view on our work. The abstraction interpretation literature distinguishes between *goal-dependent* analyses which must be performed again for every query and *goal-independent* ones which are reusable across queries. Our type system is compositional, so most of the work is reusable across queries. When a new query is provided, on the query itself (and success continuation) must be checked again. This is true in large part because procedure typechecking is static and need not know what arguments will be supplied at runtime.

6 Future Work

Our proof-of-concept implementation has shown that the certifying compilation approach is viable for logic programs. What remains is to exploit this potential by building a production-quality optimizing compiler for a widely-used language. Full Prolog is a natural target: a first step can be achieved easily by reintroducing cuts and negations as failure into the language but leaving them out of the certification spec. That is, it is straightforward to support compilation of cut and negation while only providing a formal correctness guarantee for the "pure" subgoals. It is less obvious how to certify full Prolog precisely. The deepest challenge is that provability semantics are insufficient to certify non-logical

Prolog features, so a more complex approach using operational semantics may be needed.

Logic languages other than Prolog may benefit from certifying compilation, especially certification of search soundness. Lambda-Prolog [22] and Elf [24,25] can both be easily interpreted with a provability semantics and have both been used in theorem-proving [11,26] where soundness of proof checking is essential. It is expected that these languages could be supported by using a stronger logic for specifications. Certifying compilation for Datalog might be especially fruitful given Datalog's commercial successes [2,12] and given that it is a subset of Prolog, one which typically omits cut and negation. The main challenge there would not be extending the specification language, but replacing our WAM-like design with a relational algebra-based forward-chaining interpreter as is typically used for Datalog.

The challenge of runtime performance should also not be ignored. TWAM's proximity to WAM and purely compile-time approach show promise for runtime efficiency. However, the WAM supports a well-known set of optimizations that have a significant impact in practice [1] and many of which we did not implement. Some of the most important optimizations, such as careful register allocation and common subexpression elimination, are already possible in TWAM. Many of the other important optimizations, such as jump-tables, are implemented with custom instructions, which we believe could be added to TWAM with modest effort. In short, the future work is to use the lessons learned from a proof-of-concept implementation for a simplified language to build a production-quality implementation for a production-quality language.

Acknowledgements. We thank the many collaborators and friends who read earlier drafts of this work, including Jean Yang, Jan Hoffman, Stefan Muller, Chris Martens, Bill Duff, and Alex Podolsky. We thank all of our anonymous reviewers, especially for their infinite patience with the technical details of the paper. Special thanks to the VSTTE organizers for allowing us additional space. The first author was partially supported by the NDSEG Fellowship.

References

1. Aït-Kaci, H.: Warren's Abstract Machine: A Tutorial Reconstruction. MIT Press, Cambridge (1991)
2. Aref, M., et al.: Design and implementation of the LogicBlox system. In: Sellis, T.K., Davidson, S.B., Ives, Z.G. (eds.) Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, 31 May - 4 June, 2015, pp. 1371–1382. ACM (2015). http://doi.acm.org/10.1145/2723372.2742796
3. Beierle, C., Börger, E.: Correctness proof for the WAM with types. In: Börger, E., Jäger, G., Kleine Büning, H., Richter, M.M. (eds.) CSL 1991. LNCS, vol. 626, pp. 15–34. Springer, Heidelberg (1992). https://doi.org/10.1007/BFb0023755
4. Bohrer, B., Crary, K.: TWAM: a certifying abstract machine for logic programs. CoRR abs/1801.00471 (2018). http://arxiv.org/abs/1801.00471

5. Börger, E., Rosenzweig, D.: The WAM–definition and compiler correctness. In: Logic Programming: Formal Methods and Practical Applications, pp. 20–90 (1995)
6. Bowman, W.J., Ahmed, A.: Typed closure conversion for the calculus of constructions. In: Foster and Grossman [10], pp. 797–811. https://doi.org/10.1145/3192366.3192372
7. Crary, K.: Toward a foundational typed assembly language. In: Aiken, A., Morrisett, G. (eds.) Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, 15–17 January 2003, pp. 198–212. ACM (2003). https://doi.org/10.1145/640128.604149
8. Crary, K., Sarkar, S.: Foundational certified code in the Twelf metalogical framework. ACM Trans. Comput. Log. **9**(3), 16:1–16:26 (2008). https://doi.org/10.1145/1352582.1352584
9. Crary, K., Vanderwaart, J.: An expressive, scalable type theory for certified code. In: Wand, M., Jones, S.L.P. (eds.) Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP 2002), Pittsburgh, Pennsylvania, USA, 4–6 October 2002, pp. 191–205. ACM (2002). https://doi.org/10.1145/581478.581497
10. Foster, J.S., Grossman, D. (eds.): Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, 18–22 June 2018. ACM (2018). https://doi.org/10.1145/3192366
11. Gacek, A.: System description: Abella - a system for reasoning about computations. CoRR 2008 (2008). http://arxiv.org/abs/0803.2305
12. Hajiyev, E., et al.: Keynote address: QL for source code analysis. In: Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007) (SCAM), pp. 3–16, October 2007. https://doi.org/10.1109/SCAM.2007.31
13. Harper, R., Honsell, F., Plotkin, G.D.: A framework for defining logics. J. ACM **40**(1), 143–184 (1993). https://doi.org/10.1145/138027.138060
14. Kang, J., et al.: Crellvm: verified credible compilation for LLVM. In: Foster and Grossman [10], pp. 631–645. https://doi.org/10.1145/3192366.3192377
15. Kriener, J., King, A., Blazy, S.: Proofs you can believe in: proving equivalences between prolog semantics in Coq. In: Peña, R., Schrijvers, T. (eds.) 15th International Symposium on Principles and Practice of Declarative Programming, PPDP 2013, Madrid, Spain, 16–18 September 2013, pp. 37–48. ACM (2013). https://doi.org/10.1145/2505879.2505886
16. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In: POPL 2014, pp. 179–191 (2014). https://doi.org/10.1145/2535838.2535841
17. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: Morrisett, J.G., Jones, S.L.P. (eds.) Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, 11–13 January 2006, pp. 42–54. ACM (2006). https://doi.org/10.1145/1111037.1111042
18. Martelli, A., Montanari, U.: An efficient unification algorithm. ACM Trans. Program. Lang. Syst. **4**(2), 258–282 (1982). https://doi.org/10.1145/357162.357169
19. Morrisett, J.G., Crary, K., Glew, N., Walker, D.: Stack-based typed assembly language. J. Funct. Program. **13**(5), 957–959 (2003). https://doi.org/10.1017/S0956796802004446

20. Morrisett, J.G., Walker, D., Crary, K., Glew, N.: From system F to typed assembly language. ACM Trans. Program. Lang. Syst. **21**(3), 527–568 (1999). https://doi.org/10.1145/319301.319345
21. Myreen, M.O., Owens, S.: Proof-producing translation of higher-order logic into pure and stateful ML. J. Funct. Program. **24**(2–3), 284–315 (2014). https://doi.org/10.1017/S0956796813000282
22. Nadathur, G., Miller, D.: An overview of Lambda-PROLOG. In: Kowalski, R.A., Bowen, K.A. (eds.) Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, 15–19 August 1988, vol. 2, pp. 810–827. MIT Press (1988)
23. Necula, G.C., Lee, P.: The design and implementation of a certifying compiler. In: Davidson, J.W., Cooper, K.D., Berman, A.M. (eds.) Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, 17–19 June 1998, pp. 333–344. ACM (1998). https://doi.org/10.1145/277650.277752
24. Pfenning, F.: Elf: A language for logic definition and verified metaprogramming. In: Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS 1989), Pacific Grove, California, USA, 5–8 June 1989, pp. 313–322. IEEE Computer Society (1989). https://doi.org/10.1109/LICS.1989.39186
25. Pfenning, F.: Logic programming in the LF logical framework. In: Logical Frameworks, pp. 149–181. Cambridge University Press, New York (1991). http://dl.acm.org/citation.cfm?id=120477.120483
26. Pfenning, F., Schürmann, C.: System description: Twelf—a meta-logical framework for deductive systems. CADE 1999. LNCS (LNAI), vol. 1632, pp. 202–206. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48660-7_14
27. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 151–166. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0054170
28. Pusch, C.: Verification of compiler correctness for the WAM. In: Goos, G., Hartmanis, J., van Leeuwen, J., von Wright, J., Grundy, J., Harrison, J. (eds.) TPHOLs 1996. LNCS, vol. 1125, pp. 347–361. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0105415
29. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM **12**(1), 23–41 (1965). https://doi.org/10.1145/321250.321253
30. Russinoff, D.M.: A verified prolog compiler for the warren abstract machine. J. Log. Program. **13**(4), 367–412 (1992). https://doi.org/10.1016/0743-1066(92)90054-7
31. Schellhorn, G., Ahrendt, W.: Reasoning about abstract state machines: the WAM case study. J. UCS **3**(4), 377–413 (1997). https://doi.org/10.3217/jucs-003-04-0377
32. Sørensen, M.H., Urzyczyn, P.: Lectures on the Curry-Howard Isomorphism, vol. 149. Elsevier, Amsterdam (2006)
33. Spoto, F., Levi, G.: Abstract interpretation of prolog programs. In: Haeberer, A.M. (ed.) AMAST 1999. LNCS, vol. 1548, pp. 455–470. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-49253-4_32
34. 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)
35. Warren, D.H.: An Abstract Prolog Instruction Set, vol. 309. Artificial Intelligence Center, SRI International Menlo Park, California (1983)
36. Wielemaker, J.: SWI-Prolog OpenHub Project Page (2018). https://www.openhub.net/p/swi-prolog. Accessed 28 Apr 2018

37. Xi, H., Harper, R.: A dependently typed assembly language. In: Pierce, B.C. (ed.) Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP 2001), Florence, Italy, 3–5 September 2001, pp. 169–180. ACM (2001). https://doi.org/10.1145/507635.507657
38. Xi, H., Pfenning, F.: Dependent types in practical programming. In: Appel, A.W., Aiken, A. (eds.) POPL 1999, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, 20–22 January 1999, pp. 214–227. ACM (1999). https://doi.org/10.1145/292540.292560