University of California
Los Angeles

Towards computer-verified proofs of correctness of logic-programming interpreters using derivations

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Vernon Ralph Austel

1997

1997

The dissertation of Vernon Ralph Austel is approved.

Herbert Enderton

Eli Gafni

Carlo Zaniolo

D. Stott Parker, Committee Chair

University of California, Los Angeles

1997

Table of Contents

List of Figures

List of Tables

Vita

1963	Born, Los Angeles, California, USA.
1989	B.A. (Linguistics and Computer Science), UCLA
1990	Teaching Assistant, Computer Science Department, UCLA
1990-1992	Research Assistant, Computer Science Department, UCLA
1991	Programmer, Schlumberger Laboratory for Computer Science
1992	M.Sc. (Computer Science), UCLA
1993-1995	Contractor, IBM
1995-present	Programmer/Analyst, IBM

Abstract of the Dissertation

Towards computer-verified proofs of correctness of logic-programming interpreters using derivations

by

Vernon Ralph Austel

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1997

Professor D. Stott Parker, Chair

This thesis presents sixteen increasingly refined kinds of derivations and proves that they are equivalent using the HOL proof checker; the first is close to the common one (such as given by Lloyd), while the last is expressed using abstract machine states modeled after the Warren Abstract Machine (WAM). The concept of derivation has never been refined like this in the verification literature. We use these versions of derivation in the first part of a proof of correctness by refinement stages for a simple version of the WAM. Much of the complexity of the WAM is due to details concerning the representation of data (terms and substitutions). We argue that it is simpler to refine data representation before control, and that backtracking should be refined last. Our refinement technique should be applicable to other logic-programming languages such as full Prolog.

CHAPTER 1

Introduction

This thesis investigates how to prove the correctness of realistic logic programming interpreters using a computer proof checker.

Logic programming (LP) languages pose interesting problems because they are (generally) non-deterministic. A LP interpreter does not compute a value but rather searches for solutions. LP programs do not give explicit information on how to find solutions (for the most part); it is up to the interpreter to try different alternatives, often by a simple method such as depth-first search. This is accomplished by saving (part of) its state when more than one alternative is possible and going back to such a saved state whenever the alternative being tried fails. Going back to a previous state is called *backtracking*; backtracking complicates interpreters for these languages, and has also complicated proposed correctness proofs. We show how to simplify the proof of correctness by separating the problem of backtracking from other aspects using derivations.

The particular LP language we use in our proof is pure Prolog, roughly as defined by Lloyd [13]; the target interpreter (which we do not actually reach) is the Warren Abstract Machine (WAM) [21]. Pure Prolog is a convenient language to analyze because it has clear, well-understood semantics; unfortunately, it is also of little practical use because it does not address such basic programming needs as I/O. We believe that our approach is not limited to pure Prolog; although derivations are currently only associated with pure Prolog, they could be used to define more practical LP languages.

The proof we present has been checked by the HOL proof assistant [8], which means that the likelihood of an error is very low. Creating such a proof is very demanding; no leaps of intuition are allowed, and proofs cannot be omitted because they are "obvious". This level of rigor inexorably brings hidden complications of paper proofs to the surface. Too much hidden or explicit complexity can make the implementation of a paper proof on computer impractical, even if it is considered (mainly) correct. Indeed, we used stepwise refinement of derivations not because of theoretical considerations, but rather because we were forced to in order to make the proof tractable.

The most difficult issues we faced concerned data representation, that is, representing abstract data structures (terms and substitutions) using low-level data structures. These details can be difficult if sophisticated data representation is used. If control is refined first, the complexity it adds unavoidably makes proofs concerning data representation harder, because control cannot be abstracted away. We refine data representation first, when it is simplest to deal with; this does not make subsequent proofs concerning control harder because the complexity of data representation can abstracted into two functions representing general interpreter actions, **get** and **put**.

Our final version of derivation is expressed using abstract interpreter states modeled after the WAM. Each resolution step is a list of states that an interpreter would pass through in the process of selecting a new goal and matching it with a clause head. Each state is related the the previous one by a function modeled after a WAM instruction (or an abstraction of a sequence of instructions), such as `allocate`, `get`, `call`, `proceed`, etc. It has H, B, E, P and CP registers similar to those in the WAM, an abstract representation of argument registers and the heap, and an abstract stack of stack frames and choicepoints. There is no backtrack instruction since these states are still only used to define a kind of derivation.

This thesis is divided into two parts, one concerning the general theoretical argument in favor of using derivations and one describing the actual implementation as an HOL proof in moderate detail. The general argument is presented in chapters 2 through 8 while the remaining ones (aside from the retrospective) concern the description of the HOL proof. No knowledge of theorem proving or HOL is needed to understand the first part. Some knowledge of HOL would be useful, but is not absolutely necessary, for the second part. We do not give an introduction to HOL; we refer the interested reader to [8] for a short tutorial. We assume some familiarity with Prolog, derivations and the WAM.

CHAPTER 2

How to use derivations to prove the correctness of the WAM

This chapter shows how we reduce the problem of proving the correctness of an LP interpreter to the problem of proving the equivalence of different kinds of derivations. An important benefit of this approach is that it allows one to ignore the problem of backtracking until the end of the proof. Although we refer to pure Prolog and the WAM in our discussion, we argue in chapter 7 that this approach may be applied to LP languages and interpreters in general. First we sketch what it means to prove an interpreter correct in general and then we present our technique.

2.1 A conventional proof of correctness

Proving a machine correct means proving that it does the "same thing" as a more abstract one; the abstract machine defines the semantics, and the concrete one implements the semantics efficiently.

The abstract semantics of pure Prolog is usually defined using derivations trees. A derivation is a list of resolution steps; a derivation tree is the collection of all derivations for a particular program and goal.[1] Each resolution step has

[1] Technically one must use equivalence classes of derivations.

an associated substitution, and the composition of the substitutions of the steps of a derivation is called the answer substitution. A derivation is a refutation if its last resolution step has an empty goal list. A pure Prolog computation is the list of answer substitutions of the refutation derivations of the derivation tree, enumerated in depth-first order.

Figure 2.1 shows a simple pure Prolog program that we shall use for illustration in this chapter. The derivation tree for the initial goal `a` using this program is shown in figure 2.2; the arrows in the figure indicate the two derivations, and the goal list for each resolution step is shown in a separate box (the circled numbers refer to a later figure). Figure 2.3 shows the same resolution steps, but this time the arrows indicate the depth-first traversal order.

```
a :- b(X), c(X).

b(1).

b(2).

c(1).

c(2).
```

Figure 2.1: Sample pure Prolog program

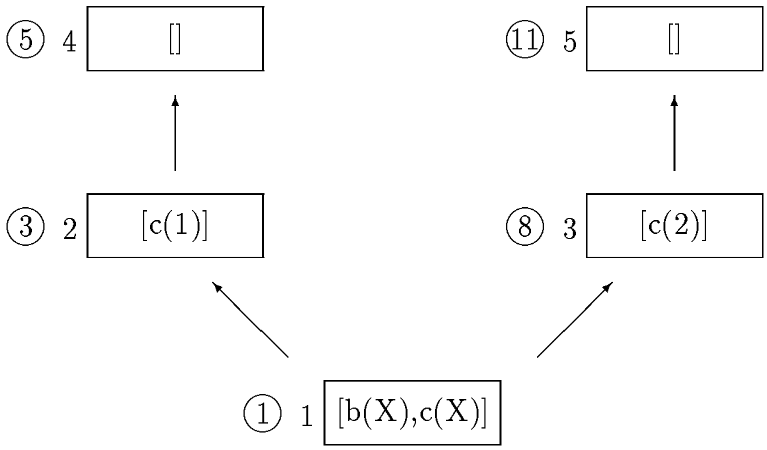

Figure 2.2: The derivation tree for goal a using the program in figure 2.1.

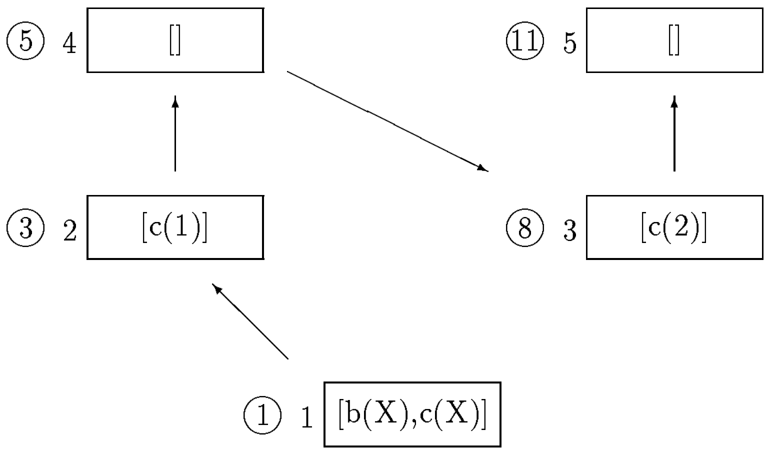

Figure 2.3: The traversal order of the derivation tree in figure 2.2.

One may also define the semantics of pure Prolog using a machine whose state is a derivation and whose state transitions correspond to the depth-first traversal of the derivation tree; during execution it either extends its derivation by another resolution step, or it pops off resolution steps until it finds the depth-first next branch of the derivation tree. We shall call this interpreter DerivInt (for derivation interpreter); the execution states of DerivInt for our sample goal and program are shown in figure 2.4.

To prove a concrete machine correct we must show that it "does the same thing" as some abstract machine. The essential idea is very simple. First one devises a correspondence relation between the abstract and concrete machine states; often abstract data structures are either explicitly represented in the concrete machine or can somehow be computed from its state. Then one shows that the initial states of the abstract and concrete machines correspond this way, and that as each of the machines run they stay in correspondence. The concrete machine often takes more than one step to get to what corresponds to the next abstract machine state.

We now illustrate this idea using an interpreter called TrivInt that is only slightly more concrete than DerivInt. Rather than formally define how TrivInt operates, we just present (in figure 2.5) the sequence of states it passes through in response to the query a for our example program. Its state contains a derivation and an integer index into the list of program clauses; the index is not defined for all states, but when it is defined it is displayed above the derivation in the figure. It has four kinds of transitions.

- call—where the machine loads the index of the first candidate program clause for resolution[2]

- get—where the machine successfully matches the head of the program clause with the first goal in the goal list and prepends the body of the body to the result of applying the unifier to the tail of the goal list

- backtrack—where the machine restarts the search for another solution after a refutation has been found.

- get/backtrack—where an attempt at resolution fails and the index of the next candidate clause is loaded.

The states TrivInt passes through are arranged to resemble the derivation tree; the circled numbers label the states in execution order, and the arrows indicate which which state comes next. States 1, 3, 5, 8, and 11 correspond to nodes in the derivation tree; the nodes in figures 2.2 through 2.4 are labelled with these numbers to make this clear.[3] Each of these states (1, 3, 5, 8 and 11) is the result of a get transition; states that do not correspond to nodes in the tree are the result of call, backtrack or get/backtrack transitions. TrivInt "does the same thing" as DerivInt because their states initially correspond and they stay in correspondence as they both execute; if one starts TrivInt in a state that corresponds to a DerivInt state, then eventually it will reach another state that corresponds to the next DerivInt state.

[2]The machine only considers those program clauses where the functor of the clause head matches the functor of current goal.

[3]The correspondence relation is very simple—the machine state *is* the DerivInt state. One could easily define a relationship that also places the other states in correspondence, but we believe that a proof of correctness for a low-level machine would not.

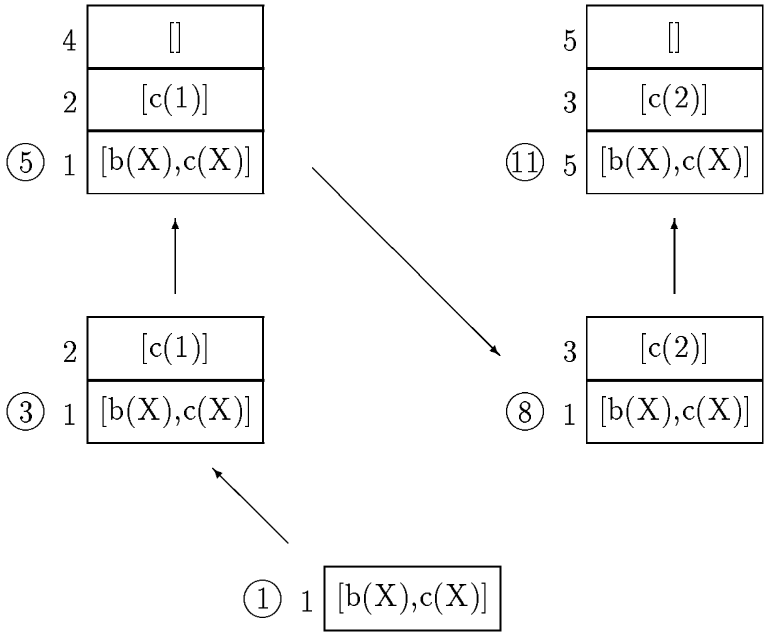

Figure 2.4: Execution of the derivation interpreter (DerivInt) for goal a using the program in figure 2.1

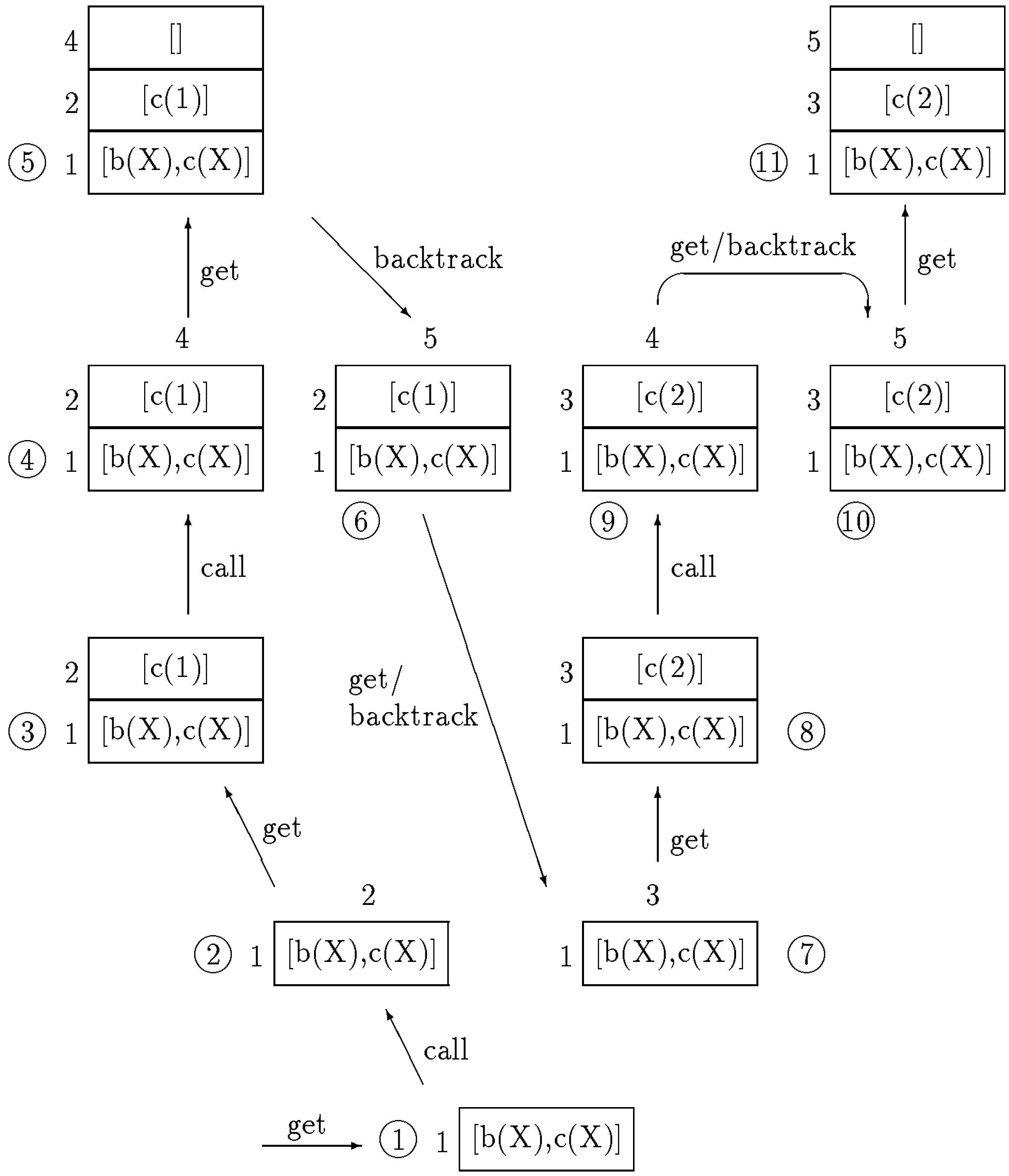

Figure 2.5: The execution sequence of TrivInt for initial goal a and the program in figure 2.1.

2.2 Derivation-oriented proof

The conventional proof style we have just sketched, which we shall call *machine-oriented*, essentially replaces the concept of derivation with an interpreter in order to have something to compare the target machine with. We propose *derivation-oriented* proof, in which we view machine execution as much as possible like a derivation, and focus on comparing different kinds of derivations.

Just as it is clear how the machine execution in figure 2.5 corresponds to the traversal of the derivation tree in figure 2.3, it should be clear that the first five states the machine passes through correspond to one of the two refutation derivations in the tree. One can construct a sequence corresponding to the other refutation derivation by changing the arrows in the diagram—one removes the backtracking transitions and adds new call transitions. Figure 2.6 shows the sequences that correspond to the two refutation derivations. One may regard these as non-deterministic execution sequences, where the machine "guesses" which clause to use at each call transition and never "makes a mistake"; in the sequence on the left, the machine guessed clauses 2 and 4, while on the right it guessed clauses 3 and 5. Not only do these execution sequences closely correspond to derivations; we shall think of the sequences *themselves* as derivations, although of a different kind.

Let the predicate is_deriv be defined as follows:[4]

---

[4]SNOC is a data constructor like CONS, but returns a list that has its first argument added to the end of its second argument rather than to the beginning. The following equations illustrate their relationship.

HD(CONS x l) = x	LAST(SNOC x l) = x
TL(CONS x l) = l	BUTLAST(SNOC x l) = l

is_deriv P [] = T

is_deriv P (SNOC x deriv) = P x deriv $\wedge$ is_deriv P deriv

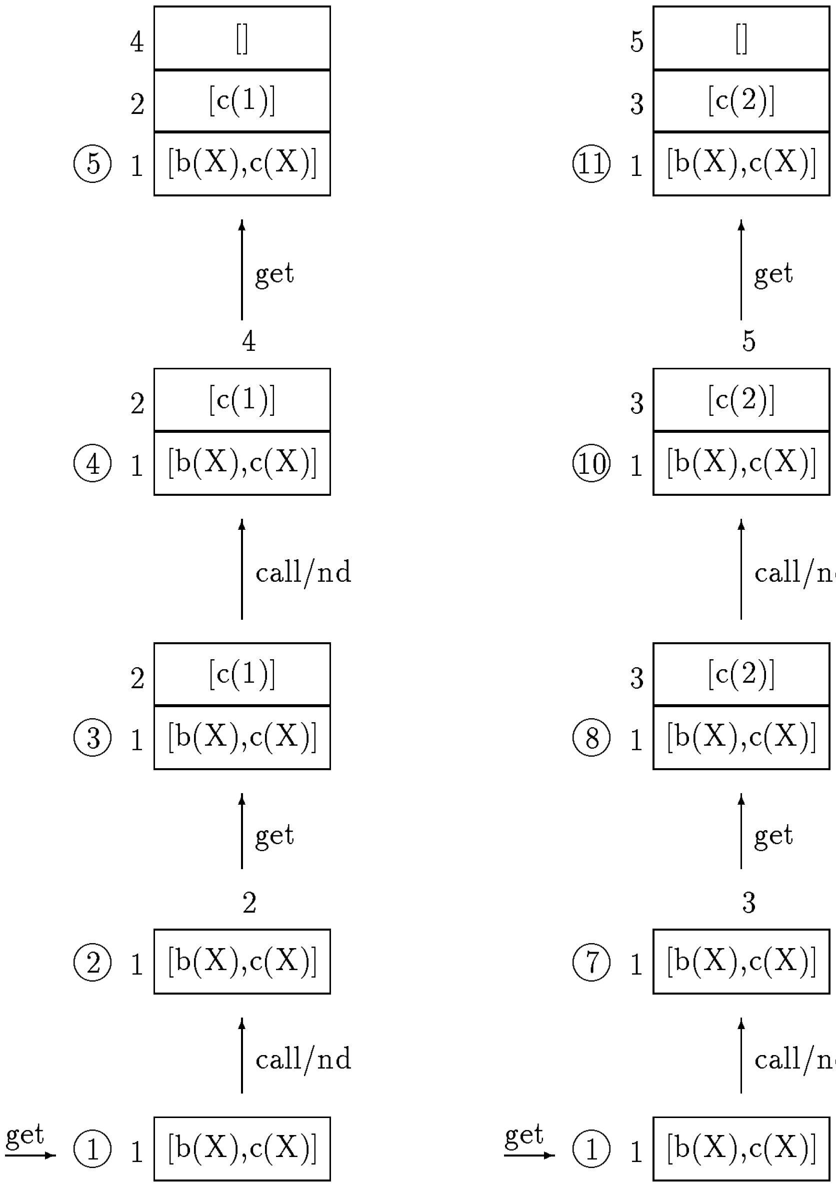

Figure 2.6: Machine-derivations of for the TrivInt machine. The circled numbers labelling machine states correspond to those in figure 2.5.

The familiar notion of derivation may be expressed as:

is_deriv is_res

where the predicate "is_res x deriv" says that x is a valid resolution step extending deriv; we shall call these *classical* derivations, and shall represent this predicate symbolically as is_classderiv.[5] If a predicate on lists can be expressed as "is_deriv P" (for some P), and lists that satisfy that predicate can be put into one-to-one correspondence with conventional derivations, then we shall call those lists a *kind of derivation*;[6] we shall also refer to the predicate itself as a kind of derivation, because it is cumbersome to refer to it as a "predicate defining a kind of derivation."

TrivInt defines a kind of derivation; the derivations are lists of lists of TrivInt states. The six derivations of this type for the initial goal a when using our example program are shown in figure 2.7. The characteristic predicate P that defines this kind of derivation must take two arguments; it returns true if the first argument contains two TrivInt states and these states are the result of executing TrivInt (for some choice of clause) starting from the last state in the last step of the second argument.[7] For example,

P [[7,8]] [[1]]

is true because the machine will pass through states 7 and 8 if it is started in state 1 and it chooses clause 3 for the call transition.

[5]For simplicity we are ignoring the fact that derivations are defined using a particular program and goal list; these would just be two more parameters to these predicates.

[6]Actually, it is only necessary to put equivalence classes of lists into correspondence with equivalence classes of derivations, since we want to allow derivations to be equivalent up to variable renaming.

[7]The derivation containing just one resolution step is an exception that has to be treated specially in every kind of derivation.

[]

[[1]]

[[1], [2,3]]

[[1], [2,3], [4,5]]

[[1], [7,8]]

[[1], [7,8], [10,11]]

Figure 2.7: The six derivations of the kind induced by TrivInt. The numbers are machine states, as given in figure 2.5.

We introduced TrivInt by showing the sequence of states passed through during execution because machine states are usually understood only with reference to execution; the machine derivations given in figure 2.6 were introduced as a kind of non-deterministic execution for the same reason. However, no concept of execution is actually required for these derivations, anymore than one is required for classical derivations. We could have defined this particular kind of derivation without ever specifying all the transitions for TrivInt; that is, we could have just defined a predicate P suitable for use with is_deriv that happens to refer to TrivInt states and then proved that is_deriv P is equivalent to classical derivations.

We now define a meta-machine that performs a depth-first traversal of derivation trees of any kind. Its state is some kind of derivation. Its two parameters are a *resolution function* that takes a derivation and a clause as input and returns a new resolution step or fails, and a *path function* that returns the list of clause numbers used by each resolution step of (this kind of) derivation.[8] We call these

[8]We are ignoring the program and the initial goal for simplicity.

the *derivation functions*. The meta-machine executes by calling the resolution function on the derivation and every program clause (in some fixed order) until one succeeds; if one does, the new resolvent is added to the derivation, otherwise it pops the last resolution step from the derivation and calls the resolution function on the popped derivation and every clause that follows the clause of the resolution step that was just popped (which can be determined using the second derivation function). It repeats this process until it succeeds or all resolution steps have been popped from the derivation. This is just a more abstract version of DerivInt.

This machine also works if we use TrivInt derivations, although the way it works in this case may seem strange. To execute, the meta-machine would call the TrivInt resolution function giving it the current TrivInt derivation and the clause to use; this function would in effect execute TrivInt to see if it successfully matches the goal, returning the list of states the machine passed through if it was successful. The state of the meta-machine is the TrivInt derivation; the states it passes through when evaluating the goal a for our sample program are the derivations given in figure 2.7.[9]

We can prove a simple but important theorem about the meta-machine. Call the sequence of clause numbers used for each resolution step in a derivation (of whatever kind) its path; the path function just returns the path.[10] The meta-machine may be thought of as enumerating the paths of a tree, the paths that its state corresponds to at each step during execution. The meta-machine will

[9]The root derivation ([]) is an exception. It seems sensible to use a state with a single resolution step as the initial state of the machine; it is convenient to represent termination as a transition to the state with the root derivation.

[10]For simplicity we are blurring the distinction between a program clause and the index into the program that identifies it.

generate a particular sequence of paths if it uses the classical derivation functions; one can show that it will produce the this same sequence no matter what kind of derivation it uses, so long as the derivation functions meet certain constraints.[11] The main constraint on the derivation functions is that the resolution function being used must succeed for a given clause if and only if the classical resolution function succeeds. If one thinks of the resolution function as generating derivations, this is the same as saying that the derivation functions must generate a derivation tree that is equivalent to that generated by the classical derivation functions. There is *no* constraint concerning backtracking, since that is implemented by the meta-machine itself.

If "DerivInt" represents the semantics of the derivation interpreter, "Machine" represents the semantics of some target machine, and "X $\sim$ Y" represents the statement "executing X is equivalent to executing Y", then the overall proof of correctness may be represented symbolically as:

$$\text{DerivInt} \sim \text{Machine}$$

"Machine" could represent something as abstract as TrivInt or something as low-level as the WAM.

We now divide this proof into three parts. We write MM(is_classderiv) to represent the execution of the meta-machine using classical derivations and the associated derivation functions; in general, if is_deriv_n is a kind of derivation then MM(is_deriv_n) represents the execution of the meta-machine using that kind of derivation and the associated derivation functions. It is simple to show that MM(is_classderiv) is equivalent to DerivInt since the meta-machine is just

[11] We are ignoring the program and the initial goal for simplicity; the sequence of paths will naturally be different for different goals and programs.

an abstracted form of DerivInt.

$$\text{DerivInt} \sim \text{MM(is\_classderiv)}$$

We then show that the meta-machine using classical derivations is equivalent to the meta-machine using machine derivations (represented by is_machderiv).

$$\text{MM(is\_classderiv)} \sim \text{MM(is\_machderiv)}$$

If we use $\equiv$ to represent equivalence between kinds of derivations, our general theorem about meta-machines says

$$is\_deriv_0 \equiv is\_deriv_1 \rightarrow MM(is\_deriv_0) \sim MM(is\_deriv_1)$$

That means that this second step just amounts to showing that classical and machine derivations are equivalent; if Machine represents TrivInt, then we would only have to prove that classical derivations are equivalent to the kinds of derivations illustrated by figure 2.6.

To complete the proof of correctness we then have to show that the meta-machine using machine derivations corresponds to the execution of the target machine.

$$\text{MM(is\_machderiv)} \sim \text{Machine}$$

Again, if Machine represents TrivInt, then one would relate TrivInt derivations (such as given in figure 2.6) to the execution of TrivInt (as illustrated in figure 2.5). At this point one would have to prove that when the backtrack instruction is executed in state 5, TrivInt will move to state 6, and that when it attempts a get transition from state 6 it will fail and backtrack into state 7; one need not reason about the transition from state 7 to 8, because that was dealt with in the second step of the proof. As figure 2.7 shows, every state resulting from a successful get transition in the machine execution occurs in the last state of the last resolution step of a TrivInt derivation, and they occur in the same order; the task of the third resolution step is to show that when the machine backtracks starting from the last state in one machine derivation, it arrives at the state that occurs in the first part of the last resolution step in the next machine derivation. The second proof step concerns success; the third proof step concerns failure.

In the case of the abstract machine we used in this chapter, the relationship between the states in the machine derivations and the states in the execution was particularly clear because they were identical. In a low-level machine like the WAM this will not be the case, but the essential ideas remain the same.

CHAPTER 3

Advantages of using derivations

The main advantage of this system becomes evident when one divides the proof process given in chapter 2 into refinement stages. Only the second of the three parts of the proof needs to be divided; one uses a sequence of increasingly complex kinds of derivations to bridge the gap between classical and machine derivations. The overall proof task looks like this.

$$\mathrm{DerivInt} \sim \mathrm{MM}(is\_classderiv)$$
$$\mathrm{MM}(is\_classderiv) \sim \mathrm{MM}(is\_deriv_0)$$
$$\mathrm{MM}(is\_deriv_0) \sim \mathrm{MM}(is\_deriv_1)$$
$$\ldots$$
$$\mathrm{MM}(is\_deriv_n) \sim \mathrm{MM}(is\_machderiv)$$
$$\mathrm{MM}(is\_machderiv) \sim \mathrm{Machine}$$

On the surface, this is just the conventional process of machine refinement; however, it is different because all proof stages but the first and last mostly concern refining derivations, not machines, since we can apply the general meta-machine theorem. The theorems needed for the intermediate stages are thus:

$$is\_classderiv \equiv is\_deriv_0$$
$$is\_deriv_0 \equiv is\_deriv_1$$
$$\ldots$$

$is\_deriv_n \equiv is\_machderiv$

Since these proofs do not deal with backtracking, one will reason about it less than one would in a conventional proof where all instructions must be dealt with. Reasoning about backtracking is not hard, but it comes up frequently and clutters up the proof. For example, in the partially refined abstract machine published by Pusch, fully half of the twelve machine transition rules involve backtracking.[1] We do not claim that using a derivation-oriented approach will save half the work involved, but it will save some, and make the proof clearer.

Second, since the meta-machine handles backtracking, it allows and encourages the proof designer to delay introducing the complex apparatus necessary to support it. The derivation equivalence proofs don't even refer to backtracking, so any complexity added to support it would serve no purpose and be pared away by someone reviewing the proof. We believe that other correctness aspects are easier to reason about when backtracking is not involved.

Third, it makes it easier to discover and understand useful invariants. We call a predicate that is true of the machine state at all points during execution an *invariant*.[2] Equivalence proofs nearly always require some kind of invariant.[3] The transitions that define the behavior of the machine do not naturally suggest what a good invariant is, and for a complex machine like the WAM they can be very difficult just to understand. Much of this complexity concerns data representation. Since we were able to develop different aspects of data representation

[1] They are: call1, call2, enter1, enter2, retry1, retry2.

[2] The word "invariant" is usually defined to be a property that holds of a machine state before and after each iteration of a loop.[9]

[3] For example, to reason about the WAM one must be able to prove that the process of dereferencing a cell eventually stops—in order to do so one proves that the store is directed. This assumes that the version of the WAM being used performs the occur-check so that circular references are never created.

separately, we were able to develop the necessary invariants incrementally, which is easier than developing a single large one to prove all aspects correct at once. Similarly, when refining forward control we did not have to deal with backtracking, so the invariants we had to develop were simpler because they not have to address that issue.

Fourth, we believe that proving the equivalence of derivations encourages modularity. A concept of derivation is just a predicate on lists. We have found it useful to divide them into conjuncts, as follows:

$$is\_deriv_n(d) = P_{na}(d) \wedge P_{nb}(d) \wedge \ldots$$

In practice, the correspondence between derivations is defined by mapping one kind to the other; we shall call this mapping function *Abs*, since we usually map the more refined kind of derivation to the more abstract one. Part of proving the equivalence of two kinds of derivations involves proving:[4]

$$is\_deriv_1(d) \rightarrow is\_deriv_0(Abs(d))$$

that is, proving that is_deriv_0 is an abstraction of is_deriv_1 means proving that for any list d that is an is_deriv_1 derivation, the function Abs maps it to a list that is an is_deriv_0 derivation. If we inline the definitions we have something like:

$$P_{1a}(d) \wedge P_{1b}(d) \wedge \ldots \rightarrow P_{0a}(Abs(d)) \wedge P_{0b}(Abs(d)) \wedge \ldots$$

We can make this task more manageable by proving individual conjuncts on the right separately.

$$P_{1a}(d) \wedge P_{1b}(d) \wedge \ldots \rightarrow P_{0a}(Abs(d))$$

[4] This is only one half of the equivalence proof, but in practice it seems to be the harder one.

$$P_{1a}(d) \wedge P_{1b}(d) \wedge \ldots \rightarrow P_{0b}(Abs(d))$$

$$\ldots$$

Finally, it is typically the case that only some of the assumptions are needed to prove particular conjuncts.

$$\begin{aligned} P_{1a}(d) \wedge P_{1e}(d) &\rightarrow P_{0a}(Abs(d)) \\ P_{1b}(d) &\rightarrow P_{0b}(Abs(d)) \end{aligned}$$

$$\ldots$$

In principle this can also be done with the invariants used in machine-oriented proof, but we believe that it is more likely to be done in derivation-oriented proof. This kind of decomposition seems to be most effective in refinements concerning data representation where control has not yet been refined, since different aspects of data representation can be refined separately, whereas control seems to affect most aspects of the machine. We believe that data representation is likely to be refined first only if derivation-oriented proof is used.

Finally, a minor advantage is that a simpler kind of induction principle is sufficient for derivation-oriented proofs. With derivations one just uses induction on a list, which is a primitive in any theorem-proving system. More complex induction principles (such as so-called rule-induction) are sometimes more cumbersome to deal with and not as well supported.

We suspect that it may be impractical to implement a computer proof of correctness for a machine as complex as the WAM if machine-oriented proof is used; the advantages we present here may mean the difference between success and failure. The author spent a year trying to mechanize the machine-oriented proof of correctness for the WAM by Russinoff [18] and finally decided that it could not be finished in a reasonable amount of time because it was too complex. Russinoff's proof does not use refinement; one might argue that Börger and Rosenzweig have overcome the problem in their proof [7] by using many refinement stages. However, we believe that the main source of complexity concerns data representation, and Börger and Rosenzweig introduce most of data representation in a single refinement stage, so the problem remains. It may be inherently more difficult to divide the data representation problem into stages using machine-oriented proofs than using derivation-oriented proofs. Most of the 16 refinement steps in our derivation-oriented proof concern data representation. This is the largest existing attempt at a proof of correctness for the WAM; attempts to mechanize Börger and Rosenzweig's proof are much smaller than ours, and do not deal with data representation issues.[5] The proof is still incomplete; although the final stage is expressed in terms of recognizably WAM-like abstract machines, and most of the data representation issues have been addressed, we believe that it would take anywhere from five to ten more stages to finish. However, at this point we see no reason why it could not be completed; we were always able to make progress on data representation problems by dividing them into different stages, and the remaining problems do not seem as difficult. In contrast, we suspect that attempts to push through a proof like Börger and Rosenzweig's may fail at the point where data representation is addressed, because too much complexity will have to introduced at once.

[5]These attempts are described in chapter 8.

CHAPTER 4

Derivations as invariants

Chapter 2 described how to decompose a proof of correctness using derivations and a meta-machine. This chapter shows this process in another light: a concept of derivation may be thought of as an invariant. First we show the role invariants have in a normal proof, and then we show the advantages of using a concept of derivation as an invariant.

Consider the general problem of trying to prove that a concrete machine implements an abstract one (or refining an abstract machine into a somewhat less abstract one). The abstract machine would have a state (call it A) and an "answer" function (say ansA) that extracts the answer from the state; the concrete machine would have similar things (C and ansC). We would then want to show that the concrete machine "does the same thing" as the abstract one. The conceptually simplest way is to just map the machine state C to machine state A (using a function called Abs, say).[1] We would like to prove something like the following:[2]

$$ansA(Abs(C)) = ansC(C)$$

Unfortunately, this is generally impossible: for non-trivial machines, the function

[1]Sometimes a more complex correspondence is required, but we shall ignore that for simplicity. It may seem odd that we choose to map concrete to abstract states rather than the other way around, like compilers do. We simply felt it to be more natural this way; the choice is in principle arbitrary.

[2]For brevity, we shall omit $(\forall C)$ from our equations.

Abs will not be defined for all possible states, so this equation simply cannot hold for all possible C.

Fortunately, it doesn't have to. We only care about certain states of our abstract machine, those through which the interpreter passes during execution. So we really only need to prove:

$$\forall n. C = stepC^n(initC) \rightarrow ansA(Abs(C)) = ansC(C) \tag{4.1}$$

where stepC is a function that describes one interpreter step, and $stepC^n$ represents recursively applying the stepC function n times.

The second thing we must prove is that the steps of the two interpreters "correspond". If the step functions for the two interpreters are called stepA and stepC, then one half of what we must prove is:[3]

$$\forall n. C = stepC^n(initC) \rightarrow stepA(Abs(C)) = Abs(stepC(C)) \tag{4.2}$$

The equality in the consequent is typically represented by a diagram like the one shown below.

$$\begin{tikzcd}
\bullet \arrow[r, "stepA"] & \bullet \\
\bullet \arrow[u, "Abs"] \arrow[r, "stepC"'] & \bullet \arrow[u, "Abs"']
\end{tikzcd}$$

We also need to prove that all abstract machines are modeled by concrete

[3] For simplicity we are assuming that the abstract and concrete interpreters progress at the same rate, that is, that no more than one concrete step is needed to correspond to one abstract step.

ones (the completeness property), that is:

$$\begin{aligned}
&\forall initA.initA\, is\ a\ valid\ initial\ state \rightarrow \\
&\quad (\exists initC.initA = Abs(initC) \wedge \\
&\quad \forall n.stepA^n(initA) = Abs(stepC^n(initC)))
\end{aligned}$$

This proof obligation is decomposed in a manner similar to equation 4.2, so we shall not consider it further.

In general, equations 4.1 and 4.2 cannot be proved as is, because the precondition $C = stepC^n(initC)$ doesn't expressly give the necessary information. Instead, we have to find a property that is maintained throughout execution and that allows us to push these proofs through, that is, we must figure out an invariant P such that:

$$\forall n.C = stepC^n(initC) \rightarrow P(C) \tag{4.3}$$

$$P(C) \rightarrow ansA(Abs(C)) = ansC(C) \tag{4.4}$$

$$P(C) \rightarrow stepA(Abs(C)) = Abs(stepC(C)) \tag{4.5}$$

Equation 4.1 follows from equations 4.3 and 4.4, and equation 4.2 follows from equations 4.3 and 4.5.

We have made no particular assumptions about the interpreter, so in general it is necessary to do this for any kind of interpreter. It is not always obvious what the invariant should be. For example, Cornelia Pusch, who proved the correctness of some simple refinements of an LP interpreter, writes in [16]:

> The predicate config_ok restricts config to admissible configurations. One of the central proof tasks is to find the right restrictions. For

each refinement step, several attempts were necessary to find the right solution.

What Pusch describes as restrictions, we would call invariants.

We believe that in the case of LP interpreters a good initial choice for the invariant P(x) is "x is a derivation". From this point on we are not talking about interpreters in general; we are specifically going to address the problem of proving the equivalence of two interpreters that both have derivations as their state. This is true of the interpreter that defines our semantics (DerivInt); we are now going to refine DerivInt in such a way that the refined interpreters will also have derivations as states. From now on, both state A and C represent (different kinds of) derivations. Our proof obligations become:

$$\forall n. C = stepC^n(initC) \rightarrow C \ is \ a \ C\text{--}derivation \tag{4.6}$$

$$C \ is \ a \ C\text{--}derivation \rightarrow ansA(Abs(C)) = ansC(C) \tag{4.7}$$

$$C \ is \ a \ C\text{--}derivation \rightarrow stepA(Abs(C)) = Abs(stepC(C)) \tag{4.8}$$

The first says that if we reach a state by deterministic execution, then we can reach it by non-deterministic execution; this is not difficult to prove. The second equation is generally also not hard to prove. The third one remains difficult.

Since an execution step is either a forward one (put/call/get) or backward one (backtrack), we can divide equation 4.8 into two parts.

$$C \ is \ a \ C\text{--}derivation \rightarrow stepA_f(Abs(C)) = Abs(stepC_f(C)) \tag{4.9}$$

$$C \ is \ a \ C\text{--}derivation \rightarrow backtrackA(Abs(C)) = Abs(backtrackC(C)) \tag{4.10}$$

Equation 4.10 is simple since both of these interpreters have a state that is a derivation: in this case, the Abs function in practice is defined by recursion on the derivation, and the backtrack operation just drops the last resolution step.

$$\begin{aligned} Abs[\,] &= [\,] \\ Abs(SNOC\ x\ deriv) &= SNOC(makeAbsStep\ x\ deriv)(Abs\ deriv) \end{aligned}$$

Since we never backtrack on an empty list (that corresponds to failure), we have:

$$\begin{aligned} & backtrackA(Abs(SNOC\ x\ deriv)) \\ &= backtrackA(SNOC(makeAbsStep\ x\ deriv)(Abs\ deriv)) \\ &= Abs\ deriv \\ &= Abs(backtrackC(SNOC\ x\ deriv)) \end{aligned}$$

Now consider equation 4.9. A deterministic forward step may be thought of as a non-deterministic forward step using a particular choice (the first one); therefore we can prove the correctness of the deterministic case if we prove it for all the non-deterministic ones.

$$C\ is\ a\ C\text{–}derivation \rightarrow stepA_{nd}^{[s]}(Abs(C)) = Abs(stepC_{nd}^{[s]}(C)) \tag{4.11}$$

Here, the argument $[s]$ represents the choice that was made at this point ("s" for "step") ; the argument is given as a list so that we may use the same notation to speak of lists of choices. In this notation, we may define the concept of derivation as follows:

$$X\ is\ a\ derivation = \exists p.X = step_{nd}^{p}(initX)$$

That is, X is reachable by non-deterministic execution for some sequence of choices p. If we substitute using this definition in the equation, we get:

$$(\exists p.C = stepC_{nd}^{p}(initC)) \rightarrow stepA_{nd}^{[s]}(Abs(C)) = Abs(stepC_{nd}^{[s]}(C)) \tag{4.12}$$

Eliminating C, we have:

$$\forall p. stepA^{[s]}_{nd}(Abs(stepC^{p}_{nd}(initC))) = Abs(stepC^{[s]}_{nd}(stepC^{p}_{nd}(initC))) \tag{4.13}$$

We can prove equation 4.13 by applying the following equation twice:

$$\forall p. stepA^{p}_{nd}(Abs(initC)) = Abs(stepC^{p}_{nd}(initC)) \tag{4.14}$$

Assume we have a function *path* for each kind of derivation such that the following three equations hold:

$$\begin{aligned} pathX(stepX^{p}_{nd}(initX)) &= p \qquad\qquad (X = A \text{ } or \text{ } C) \\ pathA(Abs(C)) &= pathC(C) \\ initA &= Abs(InitC) \end{aligned}$$

Then we can rephrase our proof obligation (equation 4.14) as follows.

$$C \textit{ is a } C\textit{–derivation} \rightarrow Abs(C) \textit{ is an } A\textit{–derivation} \tag{4.15}$$

This is true because if we substitute the definition of derivation in this equation, we get:

$$(\exists p. C = stepC^{p}_{nd}(initC)) \rightarrow (\exists p. Abs(C) = stepA^{p}_{nd}(initA)) \tag{4.16}$$

Eliminating C, we get:

$$\forall p. \exists p'. Abs(stepC^{p}_{nd}(initC)) = stepA^{p'}_{nd}(initA) \tag{4.17}$$

Equation 4.14 implies this equation, so we need only prove the converse, which we can do if we can prove that for all p, p' must be equal to p. This is true because of our assumptions about the path functions; since we may assume that equation 4.17 is true in this proof, we may choose a witness for p' and apply pathA to both sides, which gives:

$$\begin{aligned}
pathA(Abs(stepC^{p}_{nd}(initC))) &= pathA(stepA^{p'}_{nd}(initA)) \\
pathA(Abs(stepC^{p}_{nd}(initC))) &= p' \\
pathC(stepC^{p}_{nd}(initC)) &= p' \\
p &= p'
\end{aligned}$$

In chapter 2 we divided the overall correctness proof for the WAM into three parts, where the second was:

MM(is_classderiv) $\sim$ MM(is_machderiv)

By applying the meta-machine theorem, this was reduced to proving:

is_classderiv $\equiv$ is_machderiv

Classical derivations would be represented by the predicate "is an A-derivation", machine derivations would be represented by "is a C-derivation", and the statement of equivalence would be represented by "(is an A-derivation) $\equiv$ (is a C-derivation)". Most of the effort of proving this proposition involves proving equation 4.15.

The third part of the overall correctness proof was:

MM(is_machderiv) $\sim$ Machine

This *cannot* be proved in the style we have just outlined, precisely because the state of the target machine (the WAM) is *not* a derivation; one would not be able to use an invariant like "x is a derivation" to prove equations 4.3, 4.4, and 4.5. This last part would be proved in the conventional style, by induction on the number of execution steps, which involves reasoning about how the target machine implements backtracking.

CHAPTER 5

A summary of the proof stages

The refinement process described in chapter 3 assumes that many different kinds of derivations are used. This chapter gives an idea of what the various kinds of derivations may be like by briefly summarizing the stages of our proof; the stages are explained in more detail in chapters 10 and following.

5.1 Data representation refinements

The first stage (rec_deriv) is just a definition of common derivations, expressed recursively using the is_deriv predicate.

The second stage (phi_deriv) introduces a technique to represent the effect of WAM put instructions. In the abstract semantics, all variables for a new clause variant are allocated when resolution is performed. In contrast, if a variable occurs only in the clause body, the WAM may not allocate a heap cell to represent it until the first goal it occurs in is processed (by put instructions); Börger and Rosenzweig call this *on the fly initialization*. We represent this by adding a mapping called ϕ to each resolution step; we apply both the resolution unifier and the ϕ mapping to the goal atoms. Initially ϕ is an arbitrary injective mapping on variables, but over time we add constraints on it until it models the action of the WAM put instructions.

	stage name	*description*
1	rec_deriv	usual version of derivation
Data representation		
2	phi_deriv	introduce ϕ to represent WAM put instructions
3	goal_deriv	start using shared representation of goal lists
4	cphi_deriv	apply all ϕs at once before unifiers
5	trim_deriv	use only part of the unifiers (environment trimming)
6	disj_deriv	stop using unifiers in condition for fresh variables
7	put_deriv	model on-the-fly initialization of body variables
8	arg_deriv	argument registers are a special kind of stack variables
9	fix_deriv	model dereferencing by using the fixed-point of the unifiers
10	temp_deriv	temp variables in different steps need not be distinct
11	store_deriv	introduce **get** and **put** abstract functions

Table 5.1: Summary of proof stages

Forward control

12	bodyno_deriv	stop using shared representation of goal lists
13	ds_deriv	condense each resolution into a DERIV_STEP
14	op_deriv	spread out information in each DERIV_STEP into a list of DERIV_STEPs; introduce WAM-like instructions
15	pc_deriv	replace DERIV_STEP with the more realistic AWAM

Backward control

16	cp_deriv	introduce choicepoints

Table 5.2: Summary of proof stages (continued).

The third stage (goal_deriv) introduces a shared representation of goal lists. In a classical derivation, each resolution step has its own goal list to which all the relevant substitutions have been applied. In the shared representation goal lists are not represented explicitly. First, some of atoms of some of the clause bodies of the derivation (we call this list the *raw goal list*) are selected by applying a special function to the derivation; then for each atom we apply some of the unifiers and ϕ mappings. The result is the goal list of the last resolution step; the same atoms and substitutions are used, it is just expressed in an unusual fashion. Goal lists for prior resolution steps are obtained by using this process on the appropriate prefixes of the derivation.

In the third stage unifiers and ϕ mappings are applied in an interleaved order. The fourth stage (cphi_deriv) adds constraints to the unifiers and ϕ mappings so that if first the union of the ϕ mappings is applied to the raw goal atom and only after that the unifiers are applied, the result is the same as if the original technique had been used. While it is not easy to understand why, this representation is a bit closer to what the WAM does.

The WAM sometimes uses stack cells to represent variables, and it sometimes discards these cells before the stack frame containing them is popped; this technique is called *environment trimming*. The fifth stage (trim_deriv) adds constraints to model this; whereas before the entire unifier was used, from this point on if a unifier binds a stack cell that would be trimmed by the WAM, then that binding is no longer used.[1]

The classical concept of derivation requires that variables occurring in a clause used for resolution be disjoint from those occurring in the goal list. Unifiers may

[1]Naturally, the constraint is not expressed by referring to the WAM; we express it that way only for illustration.

in principle rename variables in the goal list. The sixth stage (disj_deriv) adds constraints that disallow this; the new constraints refer only to the raw goals, not to the unifiers. Furthermore, different constraints are given for heap and stack cells, since stack cells may be reused but heap cells may not.

We consider two kinds of on-the-fly initialization: stack and temp. In the case of on-the-fly stack initialization, the WAM attempts to represent a body variable using a stack cell; if this cell has not been bound (in the right way) by the time the last atom containing that variable is reached, then the WAM must allocate a heap cell for it. On-the-fly temp initialization is when the WAM does not represent the variable at all until it reaches the first atom containing it, at which point a heap cell is allocateed for it; we represent these variables by what we call *temp* variables.[2] Both of these techniques allocate heap cells, and both are represented using the ϕ mapping. In earlier stages the constraints in effect did inefficient checks to see if a ϕ needed to map a particular variable; in the seventh stage (put_deriv) we give constraints that represent the on-the-fly conditions the WAM uses, which are more efficient, and show that they can replace the old constraints.

Argument registers are used for three distinct purposes: to represent a clause variable, to temporarily point to cells and to point to the representation of the input goal. In the first case they are just a specialized kind of stack cells—those that occur in the clause head and no later than the first body atom. The eighth stage (arg_deriv) introduces argument variables by separating this special case from the rest.

Up until this point, the composition of the derivation unifiers was applied to

[2]This has nothing to do with what are called temporary variables in the WAM. A better name for these variables would have been "virtual" or "imaginary."

the raw goal atoms. In the ninth stage (fix_deriv) we add constraints so that we may apply the fixed-point of the union of the unifiers instead. The union represents the fact that unifiers are not represented individually in the WAM store; the fixed-point represents the process of dereferencing.

In the tenth stage (temp_deriv) we show that the disjointness condition for temp variables is actually not needed. Temp variables were introduced to model variables in the body that the WAM does not immediately explicitly represent; they are just a way of representing constraints on the compiler. However, the general disjointness condition applies to *all* variables. This stage shows that we can drop this condition for temp variables, essentially because they never affect each other.

Until this point, constraints concerning variables were expressed in terms of derivation data structures such as unifiers, which were just given. In the eleventh stage (store_deriv) we introduce the abstract functions **get** and **put** to create these data structures; most of the constraints on these fields are transferred to these functions, so for example given a goal and a clause (that can match the goal) the **get** function must produce a variant of the clause and also argument registers and a substitution that unifies the goal with the variant. In addition, the store is no longer computed by collecting unifiers from the entire derivation, but rather is computed incrementally at each resolution step given the old store and the new unifier.

5.2 Forward control refinements

By the twelfth stage (bodyno_deriv) all the data representation aspects we address have been refined. We used a shared representation of goal lists to simplify proofs concerning these issues. In that representation it made sense to think of the entire derivation as representing the WAM state, however, only a fraction of the information contained in the derivation is really needed. This stage prepares the way for discarding useless information by representing the body number explicitly.[3] It is noteworthy that although this is a conceptually trivial difference, the actual HOL proof was not trivial.

Stage bodyno_deriv collected all the information needed for resolution into the last resolution step (as it is in classical derivations); the thirteenth stage (ds_deriv) discards much of this information. For example, values for the argument registers in all resolution steps but the most recent are discarded, and resolution steps that do not contribute atoms to the raw goal list are dropped—resolution steps are turning into stack frames.[4] Each step of the derivation now contains something we call a *derivation step*. Each DS is related to the previous DS in one of four different ways; these four ways develop into the WAM control structure.

Stage ds_deriv has one DS data structure per resolution step; in stage fourteen (op_deriv) each resolution step contains a list of DS data structures. Each state in the list is related to the next by an instruction-like function: ds_put, ds_get, ds_call, ds_proceed, etc. There are four kinds of instruction sequences, corresponding to the four ways that successive DS data structures were related to each other in stage ds_deriv.

The DS data structure contains fields that are used in stage ds_deriv, but not in stage op_deriv, because some WAM data structures do double-duty; for exam-

[3] The bodyno is the value computed by the function referred to in stage goal_deriv; there is one bodyno for each resolution step that determines which clause body atoms will be part of the raw goal list.

[4] At this point, a derivation is a list of lists of resolution steps; we are referring to the resolution steps of the nested lists.

ple, the argument registers produced by **get** and **put** are conceptually distinct, and so must be represented using two different fields in ds_deriv, but in op_deriv we only use one of these fields, using its value in one state of the list for **get** and its value in a later state for **put**. Stage fifteen (pc_deriv) cleans this up by introducing the AWAM (abstract WAM) data structure and maps the ds_deriv operations to it. The AWAM uses a very abstract kind of program counter for control. It also implements leaf calls, that is, no stack frame is allocated for clauses that have at most one body atom. This stage finishes the development of forward control.

5.3 Backward control refinements

Our final stage (cp_deriv) begins addressing backtracking by introducing choicepoints. The control operations are modified to resemble the WAM, that is, choicepoints may be pushed before a clause is entered, and choicepoints prevent stack frames from being popped. The main result of this stage is that adding choicepoints to the stack does not disturb the execution of the machine; the choicepoints don't *do* anything yet, we just show that they don't *interfere* with anything.

CHAPTER 6

The right refinement ordering

Börger and Rosenzweig to organize their most recent proof as follows.[7]

> ... Prolog may be seen as decomposed into disjunctive, conjunctive and unification components, which corresponds nicely to WAM handling of clause selection (predicate structure), continuations (clause structure) and representation of terms and substitution.

The ordering changed slightly from the earlier version of the proof.[5]

> ... Prolog could be seen as decomposed into conjunctive, disjunctive and unificational components, ... We proceed in that order. It could have been partially permuted, by elaborating the structure of predicates before that of clauses. We would not however elaborate the structure of terms first (like Aït-Kaci ...). It seems that the conjunctive and disjunctive components of Prolog and WAM deserve an abstract explication, unburdened at first by details and constraints of term representation.

We argue that the simplest order for derivation-oriented proof is the order we chose, which is the reverse of the order Börger and Rosenzweig chose:

data representation (unificational)

forward control (conjunctive)

backward control (disjunctive)

As we point out in chapter 3, derivation-oriented proof allows one to avoid dealing with backtracking until the end, and we hold that to be an important advantage. This rules out refining backward control in the early stages. The question remaining is the why we advocate refining data representation before forward control.

First, data representation seems to be harder. In our experience, there are more issues to deal with in data representation than in forward control, some of which (such as unsafe variables) are extremely tricky; an indication of the number of issues can be seen from the fact that we required many more stages to refine data representation than forward control (chapter 5). This is not just an artifact of our proof: Börger and Rosenzweig's own proof attempt supports our belief. Seven pages in [7] describe the initial interpreter and the remaining 44 pages are evenly divided between control (both forward and backward) and data representation. Six to nine of the twelve Compiler Assumptions[1] are related to data representation, as are four of the five WAM assumptions. The rest of their proof is divided into small stages, but the one introducing data representation is sixteen pages, four times larger than the second largest stage.[2] Finally, we present the following abstract argument for the complexity of data representation.

- Proving the correctness of a Datalog interpreter would not be hard.[3]

[1]In some cases it is not clear whether the assumption involves data representation or not.

[2]Two and a half of these pages concern unification, which may be thought of as independent, and two pages introduce a formalization of terms. This still leaves this refinement step three times larger than all the others.

[3]Datalog is a LP language that does not allow variables in program clauses.

- Proving the correctness of the WAM is hard.
- The difference between a Datalog interpreter and the WAM is data representation.
- Therefore, data representation is hard.

Second, we believe that data representation is more difficult to reason about after control has been refined. As mentioned in chapter 5, we used a shared representation of the goal list while refining data representation; each goal, and hence each variable, occurs in exactly one place in the derivation. Using a shared representation means that the entire derivation is used to represent a single target machine state. When forward control was introduced, each resolution step had to contain its own copy of whatever data it needed, which represents a machine execution more closely but means that there is no longer a unique occurrence of goals or variables. From that point on, individual resolution steps correspond to target machine states (or lists of machine states). The stage that first gathered information from the entire derivation into the individual resolution steps (bodyno_deriv) seemed oddly difficult when we proved it, considering that the conceptual change is very minor. We believe that it would have been more cumbersome to state and prove the theorems of the previous stages in the style used in bodyno_deriv; it seems to be easier to reason about variables in a shared representation. Since we also believe that a shared representation cannot be used after forward control has been refined, that implies that data representation must be harder if it is refined after control.

Finally, we argue that forward control is no harder to reason about after data representation has been refined; that is, Börger and Rosenzweig are wrong in thinking that refining data representation first will burden the following refinements with details.[4] The abstract functions **get** and **put** avoid this problem. Our initial version of derivation is rec_deriv, which contains six conjuncts. All data representation issues that we model are fully addressed by stage temp_deriv, which contains 32 conjuncts, many of which are much more complex than those in rec_deriv.[5] In the following stage (store_deriv) we introduce the abstract functions **get** and **put** and transfer most of the constraints to them, with the result that only 13 conjuncts remain. Two of these conjuncts simply assert that **get** and **put** satisfy the constraints, and another two specify where their inputs come from and where their outputs go to. The most complex of the remaining conjuncts are constraints on the root resolution step; this step in effect gives the initial conditions that the interpreter must be in before it is started. The remaining conjuncts are not much harder than the ones in rec_deriv. By encapsulating the complexity of data representation in **get** and **put**, we can refine control starting from roughly the same degree of complexity as our initial version of derivation.

[4]The point is that showing that task X is easier if it precedes task Y does not demonstrate that the order X,Y is easier than Y,X, since the gain obtained by doing task X first may be more than offset by making task Y much harder.

[5]The exact number of conjuncts is fairly arbitrary, of course; we merely intend to give a rough indication of the increase in complexity.

CHAPTER 7

Beyond pure Prolog and the WAM

In this section we explain why we believe that our approach is useful for proving the correctness of backtracking non-deterministic systems in general.

7.1 Using derivations with full Prolog

Derivations are commonly used to specify the semantics of pure Prolog, but not of full Prolog, because of the so-called non-logical features of the language. We now argue that these features do not present a problem.

First of all, abstract interpreters presented for pure Prolog (such as those given in [4, 10, 3]) already have states that we would call derivations; a new choicepoint is pushed each time a new goal is resolved, and each choicepoint corresponds to a resolution step, so this list of choicepoints is a derivation.

The first language extension we consider is the cut operator. The cut prunes away part of the search tree by removing certain choicepoints from the choicepoint stack when the cut is executed; the resulting list of choicepoints is not what we think of as a derivation. We can fix that by just marking the relevant choicepoints as invalid when the cut is executed rather than removing them; when the interpreter subsequently backtracks, invalid choicepoints can finally be removed. There would be an additional constraint on derivations concerning which steps would be marked, but these constraints would depend on derivation data structures that are not refined until very late in the proof, so the cost of the additional constraint is minor. If desired, the existing formalizations could still be used as the definition of the semantics and one could just prove that this slightly different version is equivalent. The meta-machine would have to be changed, since it would need to recognize a particular goal (the cut) and treat it specially, but this does not seem difficult.

The second extension we consider is updating the program database. Pure Prolog assumes that all clauses used for resolution come from a fixed list called the program. Full Prolog also uses a program, but special goals exist that can modify it during execution. This is in fact less of a problem than cuts are. The only role the program plays in classical derivations is to restrict the choice of what clauses may be used, in order to relate the semantics to first-order logic. In our proof we almost never actually reasoned about this, it was just a minor detail that was carried along. To allow the program to be changed, one need only add a second state component (the program) to the meta-machine and make the necessary changes to its rules so that it recognizes the special goals that modify the program.

Something similar may be done for I/O. A derivation is a record of *successful computation*; when refining derivations, we are only refining how such a successful computation is represented, and not necessarily *why* is was successful. To model input, the resolution condition would be changed so that if the goal atom has a particular functor (whichever one causes a term to be read in) then *any* substitution on its variables is permitted (so long as the range contains fresh variables); to model output, if the goal atom has a particular functor (the one causing a term to be written) then the associated substitution is always the identity map. Again, the meta-machine would have to be modified to recognize these goals.

7.2 Other target machines

We used the WAM because it is a well-known and successful implementation of Prolog. Alternatives have been proposed, and different LP languages have been proposed with particular efficient interpreters in mind.[17]. We see no reason why proofs of correctness for other target machines would not benefit from our approach. We have used WAM concepts (such as its instructions and argument registers) to explain our approach, but nothing actually depends on them.

7.3 LP languages in general

We believe that the semantics of any backtracking non-deterministic LP language can be naturally expressed using derivations. An interpreter for such a language performs a search; it must sometimes explore more than one choice at a time, so it must be able to record what choices were made in order to be able to work on the alternatives at a later time. Derivations seem to be a natural means of doing this, so our technique should be applicable.

7.4 A hybrid approach

For concreteness, we described the derivation approach as proving the equivalence of the semantics with a low-level interpreter such as the WAM. In practice, one may instead opt to define an intermediate machine, prove the correctness of that machine using derivations and then continue the correctness proof of the target machine using the usual machine-oriented proof style. For example, it may turn out that our technique is best suited for the data representation refinements; one might stop at stage bodyno_deriv or ds_deriv and begin a conventional proof from there. The meta-machine approach requires a certain amount of overhead, and it may be easier to use a combination of techniques rather than one alone.

7.5 Limitations

In principle *every* computation may be thought of as a derivation, but we believe that our technique is most useful for computations with backtracking non-determinism.

Backtracking systems are interesting in that a large part of the history of a computation is *irrelevant* for its future, and this part can be easily characterized. Those steps that were backtracked out of no longer play a role; it is true that full Prolog operations such as database operations and I/O do affect the environment of the computation, but we may prove useful properties of the system that hold for all possible environments, which essentially allows us to ignore it.

Ordinary deterministic systems do not have this property; parts of the history of the computation may be irrelevant, but they cannot be easily characterized. Concurrent systems *do* have this property, and people have already made use of it by proving the correctness of processes with respect to an environment that satisfies certain conditions.[12]. Non-deterministic, non-concurrent systems that do not backtrack do *not* have this property.

We believe that derivations are especially attractive for LP because of data representation, specifically the issue of fresh variables and *re-using* variables. LP languages assume that in each step a new variant is chosen with fresh variables. This constraint is expressed very abstractly in the semantics and is generally satisfied using very complex constraints in the implementation. Derivations have the advantage of being able to express both of these styles in one system.

It may be that this approach is also useful for reasoning about exceptions in functional or imperative languages. One may view exceptions as being something like a non-deterministic construct; when doing initial refinements, one could assume that an oracle was available to determine which branch to take and thereby avoid developing a mechanism to save the state of the computation and return to it if an exception is signaled. Saving the state for an exception does not seem nearly as difficult as what is involved for a choicepoint, however, so the advantages may not outweigh the additional overhead required.

CHAPTER 8

Related work

Very little work has been done one proofs of correctness for LP interpreters. The only two published proofs are by Russinoff and by Börger and Rosenzweig. Three German researchers have worked on implementing Börger and Rosenzweig's proof on a computer; ours is the first to follow Russinoff's.

8.1 Börger and Rosenzweig

Börger and Rosenzweig have published a proof of correctness for Prolog[7][1] that also consists of many refinement stages related to each other by proof maps similar in spirit to ours. We believe that the main difference between our approach and theirs is emphasis: Börger and Rosenzweig emphasize control aspects, whereas we emphasize data representation. Control is refined in the early stages of their proof, and some of these stages have already been implemented on computer (see sections 8.5 and 8.6 below). Data representation is not treated until control has been completely refined; their goal was to be able to refine control "unburdened at first by details and constraints of term representation".

Unfortunately, this means that when data representation is developed the proof is burdened by the details of control. We believe that data representation

[1]This paper is a modified version of two earlier papers[5, 6].

is conceptually more difficult than control; Börger and Rosenzweig's own proof supports our belief, since much of data representation is introduced in a single refinement step (the Pure Prolog Theorem, on page 73), and this step is three to four times larger than any other step in the proof. This stage contains an error concerning the refinement of the abstract unification mechanism that illustrates how subtle data representation can be. The proof starts with an abstract model that uses a *rename* function to produce variants and a *unify* function to create the unifier. The Pure Prolog Theorem is proved by mapping refined interpreter states to the abstract model;[2] however, the abstract machine obtained from mapping the refined machine cannot satisfy one of the original constraints. In the abstract model, every new clause is required to use variables that have never been used before; in contrast, the refined interpreter may well use cells that have been used before to represent the variables of a new clause, since heap cells are reclaimed whenever the machine backtracks, and stack cells may be reused even if it doesn't. Since the proposed mapping function maps cells directly to variables, the abstract machine obtained through this mapping will not necessarily always use fresh variables, so it cannot satisfy the original constraint. The details are given in the appendix.

This problem is not difficult to fix. One could add a step counter to the refined machine and pass the counter to the mapping function to ensure that cells are mapped to distinct variables in different steps; one could then show in a subsequent stage that this counter is not needed for execution. Alternatively, one could define a general correspondence relation instead of a mapping function; as explained in chapter 4, one would then show that the abstract and refined

[2]Actually, it is only maps them to states from the previous refinement stage, but one could compose the refinement functions to map it all the way back to the abstract model.

interpreters start in correspondence and stay in correspondence as they progress. The point is not that these fixes add a lot of complexity—the point is that this stage is already very complex, perhaps too complex to be implemented. This may be an unavoidable consequence of refining control first.

8.2 Russinoff

Russinoff's proof[18] is both detailed and impressively accurate. It was our original goal to implement this proof; however, we were unsuccessful because beyond a certain point it became too complex for us to understand.

Russinoff uses two refinement steps: one simple one from (classical) derivations to abstract interpreter states, followed by an extremely difficult one from interpreter states to his version of the WAM. While the second step is decomposed into a number of theorems, most of them are interdependent, so one must understand how they are related in order to complete the proof, which makes it complex and difficult. We avoid this problem by dividing our proof into many refinement stages.

Proofs in the second stage must be expressed using the target machine, so abstract issues must be expressed awkwardly using low-level machine concepts. The main abstract concept is that of a term; the corresponding machine concept is the term represented by a cell. Only certain cells in a given state represent terms; these are called *objects* of the state. Reasoning about terms and substitutions represented by a state is much more tedious than just reasoning about terms, because one must always show that the cells representing the terms are objects, and one must then translate the cells into terms. We prove abstract properties using terms and substitutions directly, and then propagate the results to refinement stages. This may not actually save effort, since a certain amount of work is required to propagate the results, but it makes the proofs themselves clearer.

Börger and Rosenzweig incorrectly report that Russinoff's proof is based on a specific compiler[7]; in fact, it only assumes that the compiler satisfies certain properties, and the compiler he exhibits is only intended as an example. However, unlike Börger and Rosenzweig, Russinoff does not attempt to motivate his choice of properties.

8.3 Aït-Kaci's tutorial

Aït-Kaci has written a tutorial on the WAM[2] that presents three machines in increasing order of complexity, with some guidance as to why design decisions were made. We learned how the WAM worked mostly by studying Russinoff's proof, but Aït-Kaci's tutorial helped by giving a detailed independent implementation. It does well as a tutorial, but it was not intended as a proof, and we believe that it would not serve well as the basis for one.

8.4 Kursawe and Nilsson

Kursawe describes how to partially evaluate a Prolog interpreter to obtain a compiler[11]; this is an easier way of getting a compiler than writing it directly, since it is generally easier to write an interpreter. Nilsson extends Kursawe's work from partially evaluating the unification procedure to partially evaluating control[14].

Kursawe does not directly address the problem of correctness, but his work does indirectly concern it, since if the the interpreter used for partial evaluation and the partial evaluator itself are both correct, then the resulting compiler will be correct.[3] If the abstract interpreter is defined on the same machine that the compiler is supposed to target, and the interpreter is already suited for partial evaluation, then this completely solves the correctness problem. For example, one may define the semantics of a complex LP language (such as Flat Concurrent Prolog) using an interpreter written in Prolog, and partially evaluate Flat Concurrent Prolog programs into Prolog.[4]

In other cases, and in particular in the case of the WAM, partial evaluation is insufficient. Partial evaluation is a more-or-less automatic process; little human intervention is supposed to be required. In contrast, a great deal of human guidance is currently necessary to create a proof of correctness for a low-level interpreter that would be suitable for partially evaluating into the WAM; our proof requires 15 refinement steps to reach something that is still not quite a suitable interpreter. This is not just a consequence of today's technology. The software-development technique of partial evaluation is roughly equivalent to the verification technique called rewriting, in that both expand function calls and do straightforward expression simplification. Much of the work of proving the equivalence of interpreters involves proving properties using induction.[5] Nothing in partial evaluation corresponds to induction, so partial evaluation alone cannot do this; furthermore, it *should* not attempt to do this if it is to be a fast and

[3] For brevity we speak only of compilers; in fact, partial evaluation produces the combination of a compiler and an instruction set that defines the target machine.

[4] Here the interpreter defines the abstract semantics, Prolog defines the target machine, and the compiler produces Prolog programs given Flat Concurrent Prolog programs. In this case, the instruction set produced by partial evaluation is Prolog itself.

[5] For example, one must show by induction that when the WAM `proceed` instruction is executed then the program counter that gets loaded into the machine will point to a valid instruction.

useful software-development tool.

There is no question that partial evaluation is a useful tool for simplifying correctness proofs; we fully intended to use partial evaluation in our own proof. Unfortunately, it seems to be the case that a great deal more than partial evaluation is required for correctness proofs of low-level interpreters like the WAM.

8.5 Cornelia Pusch

Cornelia Pusch has presented four steps of a formal refinement of the WAM[16] which she implemented in the HOL logic of the Isabelle system[15]. Her work is completely formal and of high quality. However, her proof focuses on control rather than data representation, so it is difficult to compare her work with ours.

Pusch bases her proof on that of Börger and Rosenzweig[7]. Her initial attempt was in the theorem proving system KIV[20]; of this attempt she writes:

> This case study revealed that the formal proofs were significantly more involved than estimated by (Börger and Rosenzweig). For example, the correctness proof of the first refinement step used an invariant property which covered an entire page.

Pusch uses the abstract machine defined by Debray and Mishra[4] as her initial machine rather than the one used by Börger and Rosenzweig, so her first and second steps are unrelated to the ones in Börger and Rosenzweig's proof.[6] Her

[6]We think it is noteworthy that Pusch's first step is to eliminate the dump component of the machine defined by Debray and Mishra. The dump component makes the machine state a tree; she notes that it is easier to reason about the machine once this is eliminated, because then the choicepoint stack is just a list. We believe that the concept of a state dump makes sense with deterministic languages but not with LP ones.

third and fourth refinements seem roughly similar to two steps in Börger and Rosenzweig, but she does not appear to follow the proof in detail.

Since Pusch generally follows Börger and Rosenzweig's proof, her four refinement steps do not yet deal with data representation; she uses an abstract data type for substitutions so variables are not even mentioned. We fear that this proof attempt may fail at the proof stage where variables are introduced, for much the same reason that our attempt to mechanize Russinoff's proof failed. A great deal of difficult detail (16 pages worth) is introduced in this one step of Börger and Rosenzweig's proof, and this amount of complexity may simply be too great.

Finally, while her proof is impressive, ours seems to be much larger. Not only does it have four times as many stages, but judging from her description many of our stages are much more complex.

8.6 Shellhorn and Ahrendt

Gerhard Shellhorn and Wolfgang Ahrendt have also worked on formalizing Börger and Rosenzweig's proof[1, 19]. Since they are implementing the same proof, we believe that they will run into the same problems described in the preceding section on Pusch: the refinement step introducing data representation will be too complex to mechanize. They have specified six of the stages, but have only actually proved the equivalence of two (that of stages 1 and 2, and that of stages 4 and 5), neither of which involves data representation. The total proof effort for the stage 1/2 equivalence is reported to be two person-months, although a student was subsequently able to prove the necessary theorems in two weeks.

CHAPTER 9

The proof

We have attempted to present and motivate our general proof approach in the preceding chapters, which should be accessible to the general reader. The following chapters describe the HOL proof itself; although we have tried to keep the description at a high level, it is nevertheless very technical and difficult.

The HOL proof is contained in 59 files with just over 90,000 lines, written in just under a year and a half. The following sections attempt to give a general impression of what is being accomplished in each stage. These stages were not thought out on paper in advance, but rather were developed using the theorem prover; much of the time, the author had only a vague idea what needed to be done and was groping for usable constraints. Although we attempted to express the concepts as abstractly as possible, we were nevertheless forced to deal with all of the details.

While we hope that the general reader will learn something of value from them, the following chapters cannot be fully understood without some familiarity with the proof. In particular, some of the proof sketches presented here may only be comprehensible for someone reading the HOL proof, since in some cases concepts used in proofs are not introduced in the text; there are simply too many concepts that would have to be introduced and explained in order to make these proofs understandable. Aside from that, the general reader should be able to understand the introductory paragraph of each chapter, and should be able to understand the first few stages (phi_deriv, goal_deriv and cphi_deriv) in their entirety.

9.1 Division into stages

Each stage X defines a kind of derivation defined in a file named mk_X_deriv.sml, so for example there are files called mk_phi_deriv.sml and mk_goal_deriv.sml. For every adjacent pair of stages X and Y in the sequence of stages there is a file called mk_Y_2_X.sml, so for example there is a file called mk_goal_2_phi.sml. If the sequence of stage-defining files is arranged from left to right, and the stage equivalence files arranged below them, then the intent is for dependencies to flow from right to left and bottom up as in the diagram in figure 9.1. In addition to

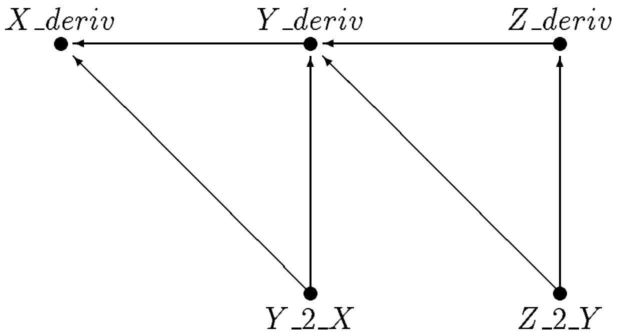

Figure 9.1: dependency graph of theories

being conceptually pleasing, this design was useful when adjustments to a X_deriv file were made. The Y_2_X files contain proofs, while the X files only contain definitions (and possibly some small proofs).[1] Consequently, it takes considerably longer to run HOL on the Y_2_X files than it does on the X_deriv files. Often, in order for a proof occurring in some stage to be pushed through, it was necessary

[1]The files X_deriv files correspond (very!) loosely to C header files, while the Y_2_X files correspond to source files.

to adjust the X_deriv file from several stages earlier; this organization permitted us to quickly re-evaluate the relevant X_deriv files and and avoid re-evaluating the intervening Y_2_X files. Eventually they had to be re-evaluated (and perhaps adjusted), but this allowed us to see if the adjustment in fact allowed us to push through the particular proof that we were working on.

9.2 Caveat concerning `mk_thm`

Unfortunately, we did not consistently hold to this model. It often occurred that there were proofs in one Y_2_X file that turned out to be useful in a subsequent one. In simple cases, we moved these proofs (and any proofs that they depended on) into the appropriate X_deriv file, however in many cases this would have involved moving more code than we wanted to. In these cases, we cheated rather than create dependencies from one Y_2_X file to another. Ordinarily one can only create HOL theorems using axioms and inference rules; this is what guarantees that any objects of type `thm` produced by the system are in fact theorems. However, HOL90 provides a function called `mk_thm` that allows one to create objects of type `thm` from arbitrary formulas. In order to avoid dependencies, we often copied a formula that we had proved correct in one Y_2_X file into a later one and created a theorem out of it using `mk_thm`. We stress that we only did this with formulas that had been proved to be theorems. The possibility exists that definitions may have been changed, and an original theorem had to be modified, but that the modification were not propagated to the copies.

CHAPTER 10

Equivalence of stages

Associated with any given program and goal is a set of equivalence classes of derivations. These are said to form a *tree*; one may enumerate these equivalence classes in *depth-first order*. An *interpretation* is the sequence of answer substitutions for the equivalence classes that contain refutations. We call a new concept of derivation *equivalent* if for any given program and goal, it also gives rise to a tree, and this tree is isomorphic to the tree induced by a classical derivation; because it is isomorphic, the depth-first traversal must produce answer substitutions in the same order. In this section, we formalize what we mean by all of these concepts.

We begin with the concept of a tree. A *path* is a list of natural numbers. Each node in a tree has a certain (finite or denumerably infinite) number of children, each of which is referred to by an index. We do not model trees with infinite paths. The path for the root node is the empty list.

Definition 1 (pathset)

```
is_tree paths =
[] IN paths /\
!path. path IN paths ==> ~(path = []) ==>
       BUTLAST path IN paths /\
       !i. i<LAST path ==> SNOC i (BUTLAST path) IN paths
```

We define depth-first order in terms of these trees; the definitions of these functions is uninteresting and is omitted. The function that performs the traversal, that is, that takes a node in a tree and returns the next node in dfs order (or the root path if there is none), is called *path_next*.

By extension, we also call a set of objects a tree if there exists a function from objects to lists of natural numbers (call it *path*) that is one-to-one on the set and such that the its image on the set is a tree. We define depth-first search on the tree of objects as follows: given an object x in the tree, return the object y in the tree such that $path\ y = path\_next(path\ \ x)$.

We define isomorphism on trees as follows.

Definition 2 (tree_iso)

```
tree_iso Pa Pb (node_patha:'a->num list) node_pathb (f:'a->'b) =
(!x. Pa x ==> Pb (f x)) /\
(!x. Pa x ==> (node_patha x = node_pathb (f x))) /\
(!y. Pb y ==> ?x. Pa x /\ (f x = y))
```

It is not hard to show that if tree_iso holds, and both the sets defined by predicates Pa and Pb are trees, and f is one-to-one on (the set defined by) Pa, then one can map the dfs traversal of tree Pa to that of tree Pb using f.

A derivation tree is a set of equivalence classes of derivations. Associate with each program clause the index where it occurs in the program. The path of a derivation is the list of clause indices used in each step; two derivations are equivalent iff their paths are identical. Therefore, one may map an equivalence class of derivations to a path, and a derivation tree to a tree.

We introduce a slight variant of tree_iso for derivation trees, since they deal with equivalence classes.

Definition 3 (settree_iso)

```
settree_iso Pa Pb (node_patha:'a->num list)
                  node_pathb (f:'a->'b) =
(!x. Pa x ==> Pb (f x)) /\
(!x. Pa x ==> (node_patha x = node_pathb (f x))) /\
(!y. Pb y ==> ?x. Pa x /\ (node_pathb (f x) = node_pathb y))
```

A trees is no longer a set of objects that satisfy a predicate (such as Pa or Pb), but rather a set of equivalence classes of objects, where two objects are equivalent if they have the same path. The function f still maps from one kind of object to another, but it is no longer required to map onto all objects of the second kind; it need only map onto some object with the same path.

We can use the same traversal function (dfs_next) for these trees. Whereas before the object it returned was the *unique* one whose path was next in the order, now it is just *any* one that is next; one then generates the node by taking the equivalence class of this object.

It is not hard to prove the following theorem.

Theorem 1 (settree_iso_dfs_next)

```
is_tree b (IMAGE pathb (SPEC Pb)) ==>
ONE_ONE_P Pb pathb ==>
settree_iso Pb Pa pathb patha (b2a:'b->'a) ==>
  !x y. Pa x ==> Pb y ==> (patha x = pathb y) ==>
            (patha (dfs_next b Pa patha x) =
             patha (b2a (dfs_next b Pb pathb y)))
```

This says that if pathb maps the set defined by Pb (`SPEC Pb`) to a tree, and pathb is one-to-one on Pb, and Pb and Pa are isomorphic (using function b2a), then (roughly speaking) b2a maps the dfs traversal of tree Pb onto tree Pa.[1] Say Pa defined classical derivations and Pb concerned a list of actual WAM states; then function b2a tells us how to recover a derivation from the machine states. What we really want is just the answer substitutions; this is in effect a function that we apply to the derivation, and when we compose this to the function b2a we would find that we only need the last machine state.

We build this b2a function that maps lists of machine states to derivations by many stages; each stage defines a refinement of the previous one and has an associated $b2a_i$, and we get the overall function by composing them.

One may show that if a tree is settree_iso to a set, then the set is also a tree. It is also easy to show that the mappings we use in our proof are one-to-one. Therefore, the main theorem in each stage is showing that settree_iso holds. It is given a name like settree_iso_Y_2_X, where Y is the name of the refined derivation type (say "goal") and X is the name of the prior derivation type (say "phi"). The main difficulty in proving this theorem is the first conjunct, namely that f maps objects of type Pb to those of type Pa; this theorem is named Y_2_X, where X and Y are as before. Much less difficult, but sometimes not easy, is the third conjunct, which is given a name like X_2_Y. Proving that f preserves paths is trivial, since the mappings we use never change the clause index of a step.

[1] It doesn't actually do this because it maps objects rather than equivalence classes of objects.

CHAPTER 11

Y_2_X: proving the mapping is into

In the last chapter we showed that proving the equivalence of each stage to the previous one boils down to proving a settree_iso theorem, which in turn depends on a theorem of the form Y_2_X, which is what requires most of the work. In this chapter, we outline how this is done.

We use stage goal_deriv as our example refinement stage; we must prove that it is equivalent to stage phi_deriv, which means that we must define a mapping (goal_2_phi) and prove that it maps any goal_deriv derivation to an (equivalent) phi_deriv derivation (equivalence is not expressed here, but is trivial to prove).

Theorem 2 (goal_2_phi)

```
!deriv. goal_deriv p g deriv ==>
          phi_deriv p g (goal_2_phi g deriv)
```

Every predicate defining a kind of derivation (such as goal_deriv) is a conjunction. At least initially, conjuncts are usually of the form `SNOC_PROP P deriv`.

Definition 4 (SNOC_PROP)

```
(SNOC_PROP P ([]:'a list) = T) /\
(SNOC_PROP P (SNOC x l) = P x l /\ SNOC_PROP P l)
```

We give the definition of phi_deriv as an example.

Definition 5 (phi_deriv)

```
phi_deriv p g deriv =
ALL_EL phi_unused_fields deriv /\
ALL_EL (clause_is_restricted_EL p) deriv /\
SNOC_PROP (clause_is_restricted_DISJOINT_phivars g) deriv /\
SNOC_PROP (phi_goals_ok g) deriv /\
SNOC_PROP (phi_mgu_ok g) deriv /\
SNOC_PROP (phi_INJ_IMAGE g) deriv /\
SNOC_PROP goal_phi_domain deriv
```

We call each of the conjuncts a *constraint*; each constraint is referred to by the predicate of interest used, so for example the fourth constraint would be called phi_goals_ok. We show its definition as an example of a SNOC_PROP predicate.

Definition 6 (phi_goals_ok)

```
phi_goals_ok g x deriv =
let goals = (deriv=[]) => [g] | RES_GOALS (LAST deriv)
in
~(goals = []) /\
(RES_GOALS x = APPEND (clause_body (RES_CLAUSE x)
                           @||| RES_THETA x)
(TL goals @||| RES_PHITHETA x))
```

Only two aspects of this constraint are relevant here: the RES_GOALS field is being constrained (since it must be equal to a particular value), and SNOC_PROP is used because this predicate needs the goal list from the previous step (LAST deriv) and it must compute the list differently for the first step (when deriv = []).

We refine derivations in the initial stages by adding or deleting constraints, or adding or deleting derivation step fields (such as RES_GOALS). For example, in stage goal_deriv we drop the RES_GOALS field and add the RES_GOAL field.

Definition 7 `(goal_deriv)`

```
goal_deriv p g deriv =
ALL_EL goal_unused_fields deriv /\
ALL_EL (clause_is_restricted_EL p) deriv /\
SNOC_PROP (goal_clause_is_restricted_DISJOINT_phivars g) deriv /\
SNOC_PROP goal_el_ok deriv /\
SNOC_PROP (goal_ok g HD_goal_deriv_goals) deriv /\
ALL_EL mgu_ok deriv /\
SNOC_PROP goal_phi_domain deriv /\
SNOC_PROP (goal_phi_INJ_IMAGE g) deriv
```

Most constraints from phi_deriv had to be changed in this stage, since most refer to RES_GOALS; only clause_is_restricted_EL and goal_phi_domain did not have to be changed. We will use goal_ok as an example of a constraint that has to replace an earlier constraint (phi_goals_ok). The reader need only remark that the predicate refers to RES_GOAL instead of RES_GOALS.

Definition 8 `(goal_ok)`

```
goal_ok g HD_goals x (deriv:res_tp list) =
(((deriv=[]) => g | HD_goals x deriv) =
        RES_GOAL x)
```

In the case of theorem 2, the constraints involved are simple enough to prove all in one theorem; generally, however, this would be infeasible. By splitting up the derivation predicate into a collection of constraints, we may easily divide the proof into subproofs, each of which concerns a single constraint. For example, if it were not so simple, we would have separately proved that goal_phi_domain is "preserved".

```
!deriv. SNOC_PROP goal_phi_domain deriv ==>
        SNOC_PROP goal_phi_domain (goal_2_phi g deriv)
```

Few constraints are so simple that they can be proved like this; in general, addition properties are generally required. For example, we could have separately proved that goal_ok "implies" phi_goals_ok [1]

```
!deriv. SNOC_PROP goal_el_ok deriv ==>
        SNOC_PROP (goal_ok g HD_goal_deriv_goals) deriv ==>
        SNOC_PROP (phi_goals_ok g) (goal_2_phi g deriv)
```

Any subset of the new derivation's constraints may be used as assumptions.

Say we have two stages, P and Q, such that $P = P_1 \wedge P2$ and $Q = Q_1 \wedge Q_2 \wedge Q_3$. We have just see that this allows us to clearly divide the overall proof that $Q \rightarrow P$ into two proofs, $Q \rightarrow P_1$ and $Q \rightarrow P_2$. It also allows us to show exactly which assumptions are needed, that is, perhaps one can show that $Q_1 \wedge Q_2 \rightarrow P_1$ and $Q_1 \wedge Q_3 \rightarrow P_2$. This allows us to assess how important particular constraints are and judge whether they should be modified. In fact, in some particularly tricky inductive cases it is probably not possible to do the proof otherwise.

In the initial stages we just add and delete derivation step fields, but in the later ones we begin representing each step first with one and then with a sequence

[1]The author has not proved this formula, so it may in fact not be a theorem, that is, more constraints may be required as assumptions.

of machine states, and we refine the machine states as we continue. Although the details are dramatically different, the general idea remains the same.

CHAPTER 12

X_2_Y: proving the mapping is onto

Proving the equivalence of a stage to a previous one requires showing that for every old-style derivation, there is an equivalent new-style derivation that is mapped onto it; by convention, this theorem has a name of the form X_2_Y. Usually there is a intuitive reason why this is true; for example, several stages just impose additional constraints on non-HEAP vars, and since every derivation has an equivalent derivation that uses only HEAP vars, so such a old-style derivation is in fact also a new-style derivation. In other stages, though, the argument is more complex. Therefore, we have used a different style that can be used for all stages.

We may state the problem in terms of the machine as follows: if a derivation exists, then the machine will find one that is equivalent to it. We have described this idea very abstractly in a predicate called `covers_all`.

Definition 9 (covers_all)

```
covers_all P PHIP HD_goals =
(!p g deriv clauseno. P p g deriv ==>
        clauseno < LENGTH p ==>
        (~(deriv=[]) ==> ~(no_goals deriv)) ==>
        ?x deriv'. PHIP p g x deriv' /\
                (RES_CLAUSENO x = clauseno) /\
```

```
                    P p g deriv' /\
                    equiv_rec_deriv deriv deriv') /\
        (!p g x deriv. P p g deriv ==>
                (~(deriv=[]) ==> ~(no_goals deriv)) ==>
                can_sld_unify ((deriv=[]) => g | HD_goals x deriv)
                        (EL (RES_CLAUSENO x) p) ==>
                PHIP p g x deriv ==>
                RES_CLAUSENO x < LENGTH p ==>
                ?x' deriv'. P p g (SNOC x' deriv') /\
                        (RES_CLAUSENO x' = RES_CLAUSENO x) /\
                        equiv_rec_deriv deriv deriv')
```

Every stage defines some notion of derivation (called something like cphi_deriv); this is passed as parameter P. The HD_goals parameter is just the function that computes the new goal.

The first conjunct says roughly that for any P-derivation that still has goals left, one can put the next goal. We will be putting constraints on the substitution called ϕ; PHIP should say something to the effect that one can always find a ϕ for the next step that will satisfy these constraints. Note that this does not say that a next step exists; it merely says that it cannot *fail* to exist because no suitable ϕ exists. This makes sense; the WAM will not backtrack during a `put` operation, only during a `get` operation.

The second conjunct says roughly that if a P-derivation exists, and there are goals left, and some clause can be unified with the next goal, then an equivalent derivation exists that can be extended by one step using that clause.

The covers_all predicate is intended to say that P-derivations cover the entire space of (classical) derivations. We do not want to have to explicitly prove that this is true for every stage. Instead, the idea is to prove that the last equivalence stage has this property; we then use the stage mapping to show that the previous stage must also have this property, and so forth down to stage phi_deriv (we explicitly show that stage phi_deriv is equivalent to stage rec_deriv). That is, at each stage we have a theorem like:

covers_all X $\rightarrow$ X_2_Y

What we really need is X_2_Y, so we must still prove that covers_all is true of stage X. Instead of doing it explicitly, we prove a theorem like the following at every stage:

covers_all Y $\rightarrow$ covers_all X

We then prove that covers_all is true of the last stage, and use these theorems to infer that it must also be true of all intermediate stages as well.

CHAPTER 13

Proof conventions

In chapter 11 we saw that constraint predicates (such as goal_ok) are typically of the form $P\ x\ deriv$, where $deriv$ is an existing derivation and x is a new element being appended to it.[1]

This pattern also holds for most of the proofs. We prove theorems of the following form by induction on the derivation.

$$SNOC\_PROP\ P\ deriv \rightarrow SNOC\_PROP\ Q\ deriv$$

Since SNOC_PROP P [] is true by definition, the new goal as the following form:

$$SNOC\_PROP\ P\ deriv \wedge P\ x\ deriv \wedge SNOC\_PROP\ Q\ deriv \rightarrow Q\ x\ deriv$$

(where we have already eliminated $SNOC\_PROP\ Q\ deriv$ from the conclusion).

Rather than write such tedious expressions, we will often say that the theorem is just

$$P \rightarrow Q$$

In proofs we will refer to x as the *new* (or *current*) step, $deriv$ as the *old* derivation, and $SNOC\ x\ deriv$ as the *new* derivation.

[1]The predicate goal_ok actually takes four parameters, but higher-order nature of HOL allows us to treat an expression like `goal_ok g HD_goals` as a predicate that takes two arguments.

In the initial stages, a derivation step consists of a set of fields which have names like RES_XYZ. The only fields used by our version of classical derivation are RES_GOALS (the goal list), RES_THETA (the unifier), RES_CLAUSE (the clause variant used) and RES_CLAUSENO (the index into the program of the clause that the RES_CLAUSE is a variant of). The goal list of the new step is therefore written `RES_GOALS x` in the HOL proofs, but this is cumbersome. In our English-language descriptions, we will simply refer to *the* goal list, *the* θ, *the* clause, etc.

The HOL data type for resolution step contains many more than these four fields, such as RES_TOPHEAP. One of the constraints in each of the stages says exactly which of the fields are actually used. The reader should be aware that as the proof progresses new fields come into use and other fields (such as RES_GOALS) are dropped. By using only some of the fields of single type rather than using distinct types for each step we were able to share important common theorems in different stages.

In some cases we need to refer to fields in prior steps; rather than use the HOL notation (such as `RES_THETA (EL i deriv)`) we will simply refer to things like θ_i, where the subscript refers to an implicit parameter *deriv*. Since we (initially) never refer to more than one derivation in a proof, this is unambiguous.

13.1 rec_deriv: classical derivations

The initial version of derivation we use is very close to Russinoff's. Each of his resolvent steps contain a goal list, a substitution, and a clause; it is required that a variant of the clause exists such that the substitution is a mgu of the variant and the appropriate goal.[2] Our version adds a new field, a clause index, and changes the meaning of the clause; instead of using a variant of the clause for resolution, we use the clause *itself*. The clause index indicates which particular program clause the resolvent step clause is a variant of. Over the course of many stages, we add conditions to the variables in this clause until it can be directly represented by data structures in the WAM.

A minor difference is that Russinoff defines derivations using three arrays and indices into the arrays, while we use a single array with fields and impose the constraints using recursion. The two styles are isomorphic.

[2]The version of derivation given by Lloyd only records the goal lists, but Russinoff's version is equivalent.

CHAPTER 14

phi_deriv: introducing the ϕ mapping

Most of the WAM instructions fall into two classes: **get** instructions (such as `get_variable`) and **put** instructions (such as `put_variable`). Each resolution step contains a unifier that may be thought of as representing (some of) the bindings created by the WAM **get** instructions when the head is matched with the goal, but the effect of the **put** instructions is not represented. This stage introduces a new field called ϕ to represent it.

We introduce a toy WAM for the purpose of this section only; for simplicity, it will not be able to backtrack. This machine is *not* used in the proof; it is *only* used to illustrate how ϕ works.

The machine has two state components: a goal and a goal stack. It has two operations: **put** and **get**. The **put** operation either transfers the top atom on the goal stack to the goal field (if the stack is not empty), or it stop the machine (if it is empty). The **get** operation selects a clause that unifies with the goal (or stops the machine if there is none), clears the goal field, and pushes the atoms in the body of the new clause onto the stack. For the moment, we will not describe how fresh variables are allocated.

Figure 14.1 shows a trivial program for this machine and an execution trace. The machine is initialized with an empty stack and the initial goal, which is

p. Execution starts with an initial **get**; it halts after step five when a **put** is attempted when the machine has an empty stack. To the far right is a corresponding derivation, where we only write the goal list for each step. With the exception of the initial one, each resolution step corresponds to two machine steps, a **put** followed by a **get**; the goal stack of the machine just after each **get** is identical to the goal list of the corresponding derivation step.

We now add variables to the program; the new program and a new execution trace are given in figure 14.2. Variables of the form H_i will represent heap cells; like the WAM, our machine will delay allocating heap cells for body variables until the first atom containing the variable is **put**. Until that point, we will represent the body variable in the goal list by just using the program variable.[1] It is no longer possible to just read off a derivation from the machine state; the unifier for steps three and four (which are not shown) cannot be used as the derivation unifier, since it binds variables H_1 and H_2, which don't occur in $h(Y)$, which would be the goal in the derivation.

The WAM **get** operation, that is, the overall effect of executing WAM **get** instructions while matching the head with the goal, is represented by a clause variant and a unifier; our derivation already has these components, and later on we shall introduce a function that produces them. We now introduce a mapping from variables to variables called ϕ to represented then WAM **put** operation, that is, the overall effect of executing WAM **put** instructions while loading argument registers with a representation of a clause body atom. In example 14.2, ϕ_1, ϕ_2, and ϕ_5 are the identity function, ϕ_3 is $Y \mapsto H_1$, and ϕ_4 is $X \mapsto H_2$. Initially, we apply ϕ to the entire goal list and require that it be one-to-one on the variables

[1]To be accurate, we would have to rename the variable if necessary, but that is irrelevant for our example.

```
p :- q, r, s.

q :- h.

h.

r.

s.
```

```
step  clause         op    goal/stack   deriv step

1     p :- q,r,s.    ---:  p []
                     get:    [q,r,s]    [q,r,s]


2     q :- h.        put:  q [r,s]
                     get:    [h,r,s]    [h,r,s]


3     h.             put:  h [r,s]
                     get:    [r,s]      [r,s]


4     r.             put:  r [s]
                     get:    [s]        [s]


5     s.             put:  s []
                     get:    []         []
```

Figure 14.1: Execution trace of the simple machine

```
p :- q, r(X), s(X).

q :- h(Y).

h(1).

r(1).

s(1).
```

```
step  clause          op    goal/stack                    deriv step???

1     p :-            ---:  p       []
       q,r(X),s(X). get:          [q,r(X),s(X)]     [q,r(X),s(X)]


2     q :- h(Y).      put:  q       [r(X),s(X)]
                      get:          [h(Y),r(X),s(X)] [h(Y),r(X),s(X)]


3     h(1).           put:  h(H_1) [r(X),s(X)]
                      get:          [r(X),s(X)]       [r(X),s(X)]


4     r(1).           put:  r(H_2) [s(H_2)]
                      get:          [s(1)]            [s(1)]


5.    s(1).           put:  s(1)    []
                      get:          []                []
```

Figure 14.2: Execution trace of the simple machine

in the goal list; later we add constraints that make it correspond more closely to the machine's behavior.

It remains to be shown that our proposed refined concept of derivation is equivalent to the original. As explained in chapter 10, this is accomplished using a tree-equivalence theorem.

Theorem 3 (settree_iso_phi_2_rec)

```
unifiable_exists_mgu ==>
(!v. atom_subvar v g ==> var_is_heap v) ==>
settree_iso (phi_deriv p g) (rec_deriv p g)
(MAP RES_CLAUSENO) (MAP RES_CLAUSENO) (phi_2_rec p g)
```

The proposition unifiable_exists_mgu is just the well-known fact that an mgu exists for any pair of unifiable terms. We assume this in order to prove (in exists_phi_heap_deriv) that for any phi_deriv derivation, there is an equivalent phi_deriv derivation that has the identity function for all ϕs. Since such a phi_deriv is in effect already a rec_deriv, that provides us with the mapping phi_2_rec. We assume that the original goal contains only HEAP vars for convenience.

If we had finished the proof, when we tied all of the stages together to prove that the final machine produces the proper answer substitutions, we would have had to have unifiable_exists_mgu as an assumption, that is, we wouldn't have our final result until we prove that it is true. This can be done using the WAM unification algorithm, since it produces a mgu for any pair of terms; we would have eliminated this assumption at the point where that algorithm would have been incorporated into the machine.

CHAPTER 15

goal_deriv: computing the goal list

The standard definition of derivation represents the goal list explicitly; that is, at each resolution step the goal list is represented by an actual list of atoms, and the next goal to be processed is simply the first atom in this list. There is no such list of atoms in a WAM interpreter. This section shows how we avoid explicitly creating goal lists; the technique we use, while actually very simple, may initially be difficult to understand, so we will use many figures to develop it. The reader *must* understand this technique in order to understand the remainder of the thesis.

There are two aspects to computing the goal list: which atoms to select, and which substitutions to apply to them. For the first part of this section we will ignore all variables and substitutions in the derivation and just show the atom functors; we shall refer to this as the *skeleton* of the derivation. The skeleton of our running example is shown in figure 15.1.

First we show the basic idea without explaining how it is done. The skeleton derivation is repeated six times in figure 15.2, each one with a line after a step, and each one showing only the goal list for the step just below the line. In each case, the goal list shown is the same as the goal list for that line in the original skeleton (figure 15.1). In this figure we introduce an arrow notation; each clause below the line is annotated with an arrow indicating which atoms in the clause body are to be used in the goal list. In each of the six cases, the concatenation (from top to bottom) of the atoms that occur after the arrows is exactly the same as the goal list. The clauses do not change, but in each case exactly one of the arrows advances, the bottommost one that is not at the end of its clause body. The goal that this topmost arrow points to is the first goal in the goal list, and hence is used for resolution in the following step. The arrow in the clause immediately below the line is always at the start of the clause body. When all arrows point to the end of their clause bodies, the derivation is a refutation. In each case, only one goal list of the derivation can be represented, but any goal list may be represented merely by properly positioning the arrows.

step	*clause*	*goal list*
6	eq.	[]
5	g.	[eq]
4	p.	[g, eq]
3	p.	[p, g, eq]
2	gp :- p, p.	[p, p, g, eq]
1	gm :- gp, g, eq.	[gp, g, eq]

Figure 15.1: How to compute the goal list from the clause bodies of the derivation (part 1). This is the skeleton of a derivation for the query gm(H_0,sue), that is, all variables and unifiers have been omitted. The original goal list would be represented here as [gm].

step	*clause*	*goal list*	*step*	*clause*	*goal list*
6	eq .		6	eq .	
5	g .		5	g .	
4	p .		4	p $\triangleright$.	[g,eq]
3	p .		3	p $\triangleright$.	
2	gp :- p, p.		2	gp :- p, p $\triangleright$.	
1	gm :- $\triangleright$ gp, g, eq.	[gp, g, eq]	1	gm :- gp, $\triangleright$ g, eq.	

Figure 15.2: How to compute the goal list from the clause bodies of the derivation (part 2). The skeleton derivation from figure 15.1 is repeated six times, except that in each case only one goal list is shown. In each case, the atoms that occur after the arrows in the clause bodies are same as those in the goal list. Due to space constraints, the six copies are displayed in this figure and the following two.

step	*clause*	*goal list*	*step*	*clause*	*goal list*
6	eq .		6	eq .	
5	g .		5	g ▷ .	[eq]
4	p .		4	p ▷ .	
3	p .		3	p ▷ .	
2	gp :- ▷ p, p.	[p, p, g, eq]	2	gp :- p, p ▷.	
1	gm :- gp, ▷ g, eq.		1	gm :- gp, g, ▷ eq.	

Figure 15.3: How to compute the goal list from the clause bodies of the derivation (part 2) (continued).

step	*clause*	*goal list*	*step*	*clause*	*goal list*
6	eq .		6	eq ▷ .	[]
5	g .		5	g ▷ .	
4	p .		4	p ▷ .	
3	p ▷ .	[p, g, eq]	3	p ▷ .	
2	gp :- p, ▷ p.		2	gp :- p, p ▷.	
1	gm :- gp, ▷ g, eq.		1	gm :- gp, g, eq ▷.	

Figure 15.4: How to compute the goal list from the clause bodies of the derivation (part 2) (continued).

In order to compute where the arrows should be placed, we introduce a new field to the derivation called *goal index*; figure 15.5 shows the skeleton derivation with the new field. Each atom in each of the goal lists comes from one of the clauses; in this figure, each atom has a subscript indicating the step in which the clause from which it was taken occurs. The goal index of step $i + 1$ is just the subscript of the first atom of the goal list for step i, that is, the goal index is the step from which the new goal for resolution was taken. Step one has no goal index, since its goal is the original goal.

We repeat the skeleton derivation six times again in figure 15.6. A new column called *sum* is added; for each step i, the *sum* entry for step i is just the number of times i appears underneath the line in the *goal index* column. For each step, the *sum* indicates where to position the arrow in the clause body; a sum of zero means position it at the beginning, a sum of one means position it after the first atom, etc.

The *goal index* and *sum* fields are determined by induction given the program clauses used in the derivation. The goal index of the step zero is never used; its sum is zero. After that, the goal index for step $i + 1$ is the largest j such that $j <= i$ and the arrow (whose position is determined by the sum field) is not positioned at the end of the clause body; the atom just after the arrow will be used as the goal. After determining the goal index, one may "raise the line" and recompute the sum column. If all arrows are at the end of their clauses, then the derivation is a refutation.

This completes our description of how to select which atoms to use for the goal list; we must now show how the proper unifiers and ϕ mappings are applied. To save space, the diagrams explaining this will use explicit goal lists, but the reader should now understand how this is implemented using the clause and goal index fields. Since we are not concerned with the actual substitutions involved, we will continue to just use the atom functor to represent the entire atom, and instead of giving an explicit unifier we will use the symbol θ_i to represent the unifier for step i.

step	*clause*	*goal list*	*goal index*
6	eq.	[]	1
5	g.	[eq_1]	1
4	p.	[g_1, eq_1]	2
3	p.	[p_2, g_1, eq_1]	2
2	gp :- p, p.	[p_2, p_2, g_1, eq_1]	1
1	gm :- gp, g, eq.	[gp_1, g_1, eq_1]	

Figure 15.5: How to compute the goal list from the clause bodies of the derivation (part 3). We introduce a new element to the derivation called the *goal index*.

step	*clause*	*goal index*	*sum*	*step*	*clause*	*goal index*	*sum*
6	eq .	1		6	eq .	1	
5	g .	1		5	g .	1	
4	p .	2		4	p $\triangleright$.	2	0
3	p .	2		3	p $\triangleright$.	2	0
2	gp :- p, p.	1		2	gp :- p, p $\triangleright$.	1	2
1	gm :- $\triangleright$ gp, g, eq.		0	1	gm :- gp, $\triangleright$ g, eq.		1

Figure 15.6: How to compute the goal list from the clause bodies of the derivation (part 4). Figure 15.5 is repeated six times, without the goal list column (these six copies are displayed in this figure and the following two). In addition, there is a sum column; in each case, for each step i, the *sum* for step i is just the number of times i appears underneath the line in the *goal index* column. Note that for each step i, the *sum* for that step tells where to position the arrow in the clause body for that step.

step	*clause*	*goal index*	*sum*	*step*	*clause*	*goal index*	*sum*
6	eq .	1		6	eq .	1	
5	g .	1		5	g $\triangleright$.	1	0
4	p .	2		4	p $\triangleright$.	2	0
3	p .	2		3	p $\triangleright$.	2	0
2	gp :- $\triangleright$ p, p.	1	0	2	gp :- p, p $\triangleright$.	1	2
1	gm :- gp, $\triangleright$ g, eq.		1	1	gm :- gp, g, $\triangleright$ eq.		2

Figure 15.7: How to compute the goal list from the clause bodies of the derivation (part 4) (continued).

step	*clause*	*goal index*	*sum*	*step*	*clause*	*goal index*	*sum*
6	eq .	1		6	eq ▷ .	1	0
5	g .	1		5	g ▷ .	1	0
4	p .	2		4	p ▷ .	2	0
3	p ▷ .	2	0	3	p ▷ .	2	0
2	gp :- p, ▷ p.	1	1	2	gp :- p, p ▷.	1	2
1	gm :- gp, ▷ g, eq.		1	1	gm :- gp, g, eq ▷.		3

Figure 15.8: How to compute the goal list from the clause bodies of the derivation (part 4) (continued).

Figure 15.9 shows the goal lists for our example derivation, where we have dropped the ϕ mappings. Recall that to compute the new goal list, one drops the first atom, prepends the body of the new clause, and applies the new unifier to the resulting list. This is what is given in the first goal list column (@ represents concatenation); steps four through six are unfortunately uninteresting in this example, since no new elements are prepended. One can easily see that by merely distributing the substitution operation over the list, one can apply all the relevant the unifiers directly to each subpart of the goal list; this is shown in the second goal list column. Here, the atoms from the new clause body are not appended to the list, but kept separate. Call the atoms in the clause bodies raw goals; clearly, if a raw goal from step i is used in the goal list for step j ($i <= j$), then the composition of the unifiers from step i to j (inclusive) is applied to it. Therefore, if we know which step the first raw goal appears in and where it is in the body (which we can do using the goal index and sum fields), then all we need to do is apply the unifiers from that step upward to it, in that order.

The situation becomes more complex when we finally take the ϕ mappings into account, in figure 15.10. To compute the new goal list, one drops the first atom, applies the new ϕ mapping, prepends the body of the new clause, and finally applies the new unifier. Again, we show two columns, one where the operations are applied straightforwardly, the other in which they are distributed across the various lists. Although the ϕs are maps from variables to variables, and hence not substitutions, we use them here as if they were (once can easily create a substitution from such a mapping). This figure is created from the previous one simply by changing all instances of θ_i to $\theta_i\phi_{i+1}$. There is a minor problem in that to maintain the pattern we have added a hypothetical ϕ_7 to step six; no such ϕ

actually exists in the derivation because it stops at this point. Note the pattern of substitutions applied to the raw goal needed for step j: if appears in step i, first apply the unifier from step i (θ_i), then apply $\phi_k\theta_k$ for $i < k < j$, then apply ϕ_j.

step	*goal list*	*goal list*
6	$[]\theta_1\theta_2\theta_3\theta_4\theta_5\theta_6$	$[]\theta_1\theta_2\theta_3\theta_4\theta_5\theta_6$
5	$[\mathrm{eq}]\theta_1\theta_2\theta_3\theta_4\theta_5$	$[\mathrm{eq}]\theta_1\theta_2\theta_3\theta_4\theta_5$
4	$[\mathrm{g},\ \mathrm{eq}]\theta_1\theta_2\theta_3\theta_4$	$[\mathrm{g},\ \mathrm{eq}]\theta_1\theta_2\theta_3\theta_4$
3	$([\mathrm{p}]\ @\ ([\mathrm{g},\ \mathrm{eq}]\theta_1))\theta_2\theta_3$	$[\mathrm{p}]\theta_2\theta_3\ @\ [\mathrm{g},\ \mathrm{eq}]\theta_1\theta_2\theta_3$
2	$([\mathrm{p},\ \mathrm{p}]\ @\ ([\mathrm{g},\ \mathrm{eq}]\theta_1))\theta_2$	$[\mathrm{p},\ \mathrm{p}]\theta_2\ @\ [\mathrm{g},\ \mathrm{eq}]\theta_1\theta_2$
1	$[\mathrm{gp},\ \mathrm{g},\ \mathrm{eq}]\theta_1$	$[\mathrm{gp},\ \mathrm{g},\ \mathrm{eq}]\theta_1$

Figure 15.9: How to compute the goal list from the clause bodies of the derivation (part 5).

step	*goal list*	*goal list*
6	$[]\theta_1\phi_2\theta_2\phi_3\theta_3\phi_4\theta_4\phi_5\theta_5\phi_6\theta_6\phi_7$	$[]\theta_1\phi_2\theta_2\phi_3\theta_3\phi_4\theta_4\phi_5\theta_5\phi_6\theta_6\phi_7$
5	$[\mathrm{eq}]\theta_1\phi_2\theta_2\phi_3\theta_3\phi_4\theta_4\phi_5\theta_5\phi_6$	$[\mathrm{eq}]\theta_1\phi_2\theta_2\phi_3\theta_3\phi_4\theta_4\phi_5\theta_5\phi_6$
4	$[\mathrm{g},\ \mathrm{eq}]\theta_1\phi_2\theta_2\phi_3\theta_3\phi_4\theta_4\phi_5$	$[\mathrm{g},\ \mathrm{eq}]\theta_1\phi_2\theta_2\phi_3\theta_3\phi_4\theta_4\phi_5$
3	$([\mathrm{p}]\ @\ ([\mathrm{g},\ \mathrm{eq}]\theta_1\phi_2))\theta_2\phi_3\theta_3\phi_4$	$[\mathrm{p}]\theta_2\phi_3\theta_3\phi_4\ @\ [\mathrm{g},\ \mathrm{eq}]\theta_1\phi_2\theta_2\phi_3\theta_3\phi_4$
2	$([\mathrm{p},\ \mathrm{p}]\ @\ ([\mathrm{g},\ \mathrm{eq}]\theta_1\phi_2))\theta_2\phi_3$	$[\mathrm{p},\ \mathrm{p}]\theta_2\phi_3\ @\ [\mathrm{g},\ \mathrm{eq}]\theta_1\phi_2\theta_2\phi_3$
1	$[\mathrm{gp},\ \mathrm{g},\ \mathrm{eq}]\theta_1\phi_2$	$[\mathrm{gp},\ \mathrm{g},\ \mathrm{eq}]\theta_1\phi_2$

Figure 15.10: How to compute the goal list from the clause bodies of the derivation (part 6).

One can see why this is by considering how the WAM works. The ϕ represents last-minute adjustments made by **put** instructions that occur before unification is attempted; these adjustments do not affect the clause body of the new clause, which is why ϕ_i is never applied to the clause body of step i. After this, the raw goal undergoes substitution by both ϕ_k and θ_k for all intervening levels, but it of course is only affected by ϕ_j just before it is unified with the new clause head.

We have seen that the goal list is constructed by taking the parts of the clause bodies after the arrows, applying certain substitutions to them, and concatenating the result. We call the part of the clause body behind the arrow of step i $body_i$. We call the list of non-nil bodies in the derivation the *stack*, and the list of atoms obtained by flattening the reversed stack the *raw goals*. The stack is the germ of the runtime stack in the WAM. Every time a new goal is computed, we remove the first raw goal, that is, the first atom in the last stack element; we refer to this as *popping the stack*. Adding a new step involves popping the old stack and adding a new element.

As we have seen, the arrows in our diagrams are just a way of displaying where the body starts; they represent a function from natural numbers to natural numbers which we will call *deriv_bodyno*.

Each goal atom is obtained by applying a substitution to a raw goal atom; we refine this substitution over several stages until we get something we think of as the WAM store. To define the substitution we use in this stage we use a function called *top_phithetas*; given i, it returns $\phi_i\theta_i \ldots \phi_n\theta_n$, where n is the index of the last step of the old derivation.

The formula we now use to compute the goal to be unified is now:

(hd $body_i$) $\theta_i \theta_{i+1} \phi_{i+1} \cdots \theta_n \phi_n \ \phi$

Here, n is the index of the last step of the derivation, and i is the index of the top of the stack, that is, it is the last step in the deriv that still has atoms left to be processed. We introduce a new field, RES_GOAL_EL, to hold the value i; it is not strictly needed, but is a convenient shorthand. The constraint that says that RES_GOAL_EL has this value for every step of the derivation is called `goal_el_ok`. A constraint called `goal_ok` is introduced; it's definition follows.

Definition 10 (`goal_ok`)

```
goal_ok g HD_goals x (deriv:res_tp list) =
(((deriv=[]) => g | HD_goals x deriv) = RES_GOAL x)
```

This constraint is used for several stages as we refine the method of computing the next goal. In all cases, the goal for the first step is just the original goal g; otherwise, a function parameter (HD_goals) is applied to the new step and the old derivation. As we introduce new methods for computing the goal, we use the same constraint, but with different values for HD_goals.

CHAPTER 16

cphi_deriv: collecting the ϕ mappings

To review, a `goal_deriv` derivation computes the goal atom to use for resolution as follows: it determines the step (say i) where the next goal atom to process comes from, applies the unifier of the ith step to it, then every ϕ and unifier after the ith step up to (but not including) the new step, and finally applies the ϕ mapping of the new resolution step. This does not resemble the way a WAM operates at all, but it is a little closer than standard derivations, since it does not compute the entire goal list.

The purpose of the ϕ mappings is to abstractly model the way the WAM allocates heap cells in a delayed fashion. The way `goal_deriv` does it is unsatisfactory, because there is nothing corresponding to the chain of alternating θ and ϕ substitutions; it would be better if all the ϕs were applied first (corresponding to the allocation operations), and then all the θs (corresponding to dereferencing the WAM store). Furthermore, we argue that it will closely model the WAM if the various ϕ substitutions are applied in parallel, not composed together like the unifiers are. This means that we want to impose conditions so that the following equation is true:

$$g\theta_i\phi_{i+1}\theta_{i+1}\cdots\phi_{n-1}\theta_{n-1}\phi_n = g\bigcup\{\phi_{j_1},\cdots\phi_{j_k}\}\theta_i\theta_{i+1}\cdots\theta_{n-1} \tag{16.1}$$

where j_i through j_k is a subset of $i+1\ldots n$, that is, not all the ϕs are being used on the right hand side. It will probably not be apparent at this point why the second form is more desirable; we simply assert that it is. The object of this refinement is to add conditions that make this equation true.

The goal_deriv stage introduced a constraint called goal_ok which takes a parameter HD_goals; this constraint says that every goal in the derivation is computed using HD_goals. We proved that goal_deriv is equivalent to classical derivations if HD_goals is HD_goal_deriv_goals. We now define a new function, HD_cphi_deriv_goals, that computes the goal as described above, and wish to show that if the goal_ok constraint is passed this function, then we still end up with something equivalent to classical derivations. In order to do so, we must impose several other constraints.

We will review a few features of the way the WAM works so that the reader may better understand why the constraints we will now impose model it. The WAM allocates heap cells to represent variables that only occur in the body. In one case it first uses a stack cell to represent a variable; while processing the last atom the variable occurs in, it may determine that it must replace uses of the stack cell by a new heap cell (just how and why this is done does not yet concern us). In the other case, it unconditionally allocates a fresh heap cell for a variable when the first atom containing it is processed. In the first case, we model the stack cell used by the WAM with a STACK variable in the clause variant of the derivation; in the second case, we use a TEMP variable to be a placeholder for the variable. In both cases, the variable gets mapped to a HEAP variable by ϕ. As we shall see, we do not yet need to distinguish between the two cases at this stage, so our first constraint is this.

Constraint 1 (cphi_heap_vars (part 2)) *No HEAP vars may occur in the domain of a ϕ, and only HEAP vars may occur in the range of a ϕ.*

The WAM compiler always ensures that heap cells point to other heap cells (or to structures that are built in the heap). The effects of unification are represented by the derivation unifiers. It is therefore reasonable to require the following constraint:

Constraint 2 (cphi_heap_vars (part 1)) *If θ maps a HEAP var v, then all subvars of θv must be HEAP vars.*

Given these two constraints, we can see that all ϕ mappings may be dispensed with entirely as far as HEAP variables in the raw goal are concerned. This follows easily by induction on the length of the list of substitutions being applied. Given a term containing only HEAP variables, the result of applying a unifier to it results in a term containing only HEAP variables, and such a term must be unaffected by the application of a ϕ mapping. This means we can drop all ϕ mappings from the left side of equation 16.1. If we apply all ϕs in parallel to a HEAP variable, none of them will apply, so we may also drop all the ϕ mappings from the right side of equation 16.1, so it holds.

We now consider non-HEAP variables that occur in the raw goal (represented by g in equation 16.1). Our first concern is to add constraints to allow us to drop those ϕ mappings that we consider irrelevant.

When the WAM pushes a new stack frame and processes the input goal, it is allocating fresh stack cells for the new clause, that is, these variables do not occur in any other goal that is still on the stack. The list of derivation steps that contain raw goal atoms yet to be processed corresponds roughly to the WAM stack. Therefore, it makes sense to impose the following constraint:

Constraint 3 (stack_disjoint) *The variables in the new clause must be disjoint* *from the variables in the stack of the old derivation after popping it.*

This means that a variable may not occur in two different levels in the stack. This is different from the original disjointness condition for derivations; that required new variables to be disjoint from the variables in the goal list. It is quite possible that some of the variables in the stack are renamed away by the unifiers or ϕ mappings so that they no longer exist in the goal list, which means that the old constraint would allow them to appear in the new clause.

When the WAM processes a body atom, it may allocate heap cells for variables that occur in that atom, but it does so *only* for variables that occur in the atom it is currently processing. This justifies the following constraint:

Constraint 4 (cphi_phi_domain 2) *The variables in the domain of the ϕ must occur in the raw goal.*

We shall say that a ϕ_i in the old derivation is *relevant* if $goal\_index_i$ is equal to the new goal_index, otherwise it is *irrelevant*; only relevant ϕs are included on the right side of equation 16.1. The two constraints we just added make the following equation true.

$$g \bigcup \{\phi_{j_1}, \cdots \phi_{j_k}\} = g \bigcup \{\phi_i, \cdots \phi_n\} \tag{16.2}$$

Let i be the new goal index (the index of the step from which the goal for the new step has been taken). Consider ϕ_j, where $i < j$, and let k be $goal\_index_j$. Then $i <= k$, because the goal index is always the largest index that has unprocessed body atoms, and step i has unprocessed goals now (since the new goal comes from step i), so it must also have had unprocessed goals when step j was new. Since we are considering the case where $i \neq k$, we now have $i < k$. When step k

was new our constraint 3 would have demanded that all the variables in $clause_k$ be distinct from variables that were on the stack at that time. The new goal is in step i, it is on the stack now, and $i < k$, so therefore it must have been on the stack when step k was new; therefore the variables in $clause_k$ must be disjoint from the variables in the new goal. Since ϕ_j is restricted to variables that occur in some atom from step k, it must be undefined for all variables occurring in the new goal atom, so we can drop it. $\Box$.

We now add a constraint to ensure that the domains of the relevant ϕs are pairwise disjoint and and are disjoint from the domain of the new ϕ.

Constraint 5 (cphi_welldefined) *If ϕ is defined for v, then all relevant ϕs are undefined for v.*

This corresponds to what the WAM does; it allocates a new heap cell for a variable the first or the last time it sees it, but it never does so twice. Obviously this ensures that the domain of the new ϕ is disjoint from the domains of all relevant ϕs. One can easily see by induction that they are also pairwise disjoint; the first relevant ϕ is trivially disjoint from the empty set of relevant ϕs, and as each new ϕ iss added, it is forced to be disjoint to the existing set of relevant ϕs.

We now continue trying to make equation 16.1 hold by considering non-HEAP variables that occur in the raw goal. There are three subcases: both ϕ and all relevant ϕs are undefined for v, ϕ is defined for v (and hence all relevant ϕs must be undefined for v), and ϕ is undefined for v but some relevant ϕ is.

The first case is handled by adding the following constraint:

Constraint 6 (subvars_unbound_phi) *If ϕ is undefined for v, and v occurs in the raw goal, and every relevant ϕ in the old derivation is undefined for v, then* *for all j where $i < j <$ length deriv, ϕ_j is unbound for all subvars of $(\theta_i \cdots \theta_j)v$, where i is the new goal_index.*

This may be thought of as giving a condition for when ϕ *is allowed* to be undefined for a variable. It is actually very simple. It says that ϕ_{i+1} must be undefined for all variables occurring in $\theta_i v$, ϕ_{i+2} must be undefined for all variables occurring in $(\theta_i \theta_{i+1})v$, etc. In other words, $(\theta_i \phi_{i+1} \theta_{i+1} \cdots \phi_{j+1})v$ must be equal to $(\theta_i \theta_{i+1} \cdots \theta_j)v$, which means we can drop the ϕs from the left side of equation 16.1. Since by assumption none of the relevant ϕ_j are defined for v, we may also drop them from the right side, and so it holds.

If the new ϕ is defined for v, we want the new HEAP variable to be "fresh". At this time we will model this concept with the following constraint.

Constraint 7 (cphi_phi_range (part)) *If ϕ maps a non-HEAP v to a HEAP var hv, then hv may not be in the new or any prior clause, it may not be in the range of any prior ϕ, and it may not be in the domain or range of any prior θ.*

Thus, if the new ϕ is defined for v, then the right side of equation 16.1 just evaluates to ϕv. The following constraint forces the left side to also evaluate to ϕv. We do this in two steps. First, we ensure that θ_i does not map v.

Constraint 8 (cphi_phi_range (part 2)) *If v is in the domain of ϕ, then $theta_i$ must be undefined for v, where i is the new goal index.*

The subsequent steps are handled by this constraint.

Constraint 9 (subvars_bound_phi) *If ϕ maps a non-HEAP var v to HEAP var hv, then for all j from i to length deriv, $(\theta_i \ldots \theta_j)v$ is a variable, $\phi((\theta_i \ldots \theta_j)v) = hv$, and ϕ_j is undefined for $((\theta_i \ldots \theta_j)v)$*

This constraint is also actually simple. At each step j we start with $v_{j-1} = (\theta_i \ldots \theta_{j-1})v$, which we may assume is a non-HEAP variable; $v_i = v$. The new constraint disallows ϕ_j from mapping v_{j-1}; it allows θ_j to map v_{j-1}, but the result must be a variable (which we will call v_j), and forces ϕ to map v_j to ϕv. Since ϕ maps v_j, it may not be a HEAP var. This proves that the left side of equation 16.1 is equal to ϕv, so it holds. Intuitively, what is happening is that the θs are allowed to map v to other non-HEAP vars, giving a string of them, but none of them may be mapped by a prior ϕ, and ϕ must map all of them to the same HEAP var. This is an very abstract model of the condition under which a stack cell is processed by the WAM put operation.

The third and final case we must consider is where ϕ is undefined for v, but some relevant ϕ is. We may assume that our equation holds for the step containing the first relevant ϕ that is defined for v; call this step j, and call the value computed by both sides of equation 16.1 at this point hv (it must be a HEAP var). On the left side we get: $(\phi_{j+1}\theta_{j+1} \ldots \phi_{n-1}\theta_{n-1}\phi)hv = \theta_{j+1} \ldots \theta_{n-1})hv$ since hv is a HEAP var, and no ϕ may be defined for HEAP vars, and all θs must map HEAP vars to terms containing only HEAP vars. On the other hand, $hv = (collect\_cphi[\phi_i \ldots \phi_j]v)\theta_i \ldots \theta_j = (collect\_cphi[\phi_i \ldots \phi_n]v)\theta_i \ldots \theta_j$[1] since $\phi_{j+1} \ldots \phi_n$ and ϕ must be undefined for v, so the right side also evaluates to $(\theta j + 1 \ldots \theta_{n-1})hv$, so the equation holds.

[1] collect_cphi selects the relevant ϕs from its input.

CHAPTER 17

trim_deriv: trimming stack vars

The fundamental difference between the WAM stack and heap is that data on the heap is never deallocated, while data on the stack eventually is.[1] At the very latest, a stack frame will be deallocated after its entire body has been processed and control is about to be returned to its caller.[2] Derivation unifiers are analogous to part of the WAM store. When a stack frame is released, the WAM store is in effect trimmed. This section adds constraints so that only part of the derivation unifiers are used; the parts that are not correspond to the variable-representing stack cells in the WAM that have been deallocated.

We review this optimization in terms of the machine first; the reader unfamiliar with this topic should refer to Ait-Kaci[2] for details.

Consider the following trivial program.

```
p :- q(X), r(X), t.
q(1).
r(1).
```

Consider two possible executions for the goal p. In the first, the machine allocates

[1]Backtracking releases data on the WAM heap, but that involves re-starting a derivation on a new path, and we are only concerned with the behavior of the machine along a single derivation.

[2]It won't if it is protected by a choicepoint, but we don't have to consider that complication at this point.

a heap cell for X, calls q, which binds the heap cell to 1, calls r, which succeeds, and then calls t; for the duration of the execution of t (in fact, for the remainder of execution) this heap cell remains in use. In the second possible execution, the machine uses a stack cell to represent X; the call to q binds the stack cell to 1, the call to r succeeds, and before t is executed the machine *deallocates* the stack cell, so that exactly as many resources are available for the computation of t as there were for the computation of p. The difference is that in the first case a resource is permanently allocated, and in the second a resource is not, allowing the machine to reuse the resource to continue the computation.

Note that when the goal $r(X)$ is processed in this example, the stack cell representing X is already bound. When the machine **put**s the goal $r(X)$, it can dereference the stack cell and pass the value 1 rather than the stack cell X to the procedure r. Thus, the rule is that we discard a stack cell just after we **put** the last atom that contains the variable represented by the cell.

This leads to the question of what happens if the cell is *not* bound by procedure q, as in the following

```
p :- q(X), r(X).
q(X).
```

The technique we are formalizing requires that the stack cell be released before the last atom containing it is processed, so we cannot pass the stack cell; instead, we must pass r a new (unbound) heap cell to represent X.

Finally, we must be careful what we bind stack cells to. Consider the following program.

```
p :- q(X,Y), r(X), s(Y).
```

```
q(X,X).
```

Say both X and Y are being represented by stack cells, and say q ends up binding Y to X. Since X is released before Y is, Y will have an undefined value when s is executed. To ensure that this situation does not arise, the WAM uses a special case of this general rule.

Bind x to y only if y will be deallocated no sooner than x.

We now describe how we formalize this optimization using derivations. We must first model stack cells.

Definition 11 *A* stack var *of a derivation is a variable that occurs in its stack.*

Recall that the derivation stack is the germ of the WAM stack; although cells on the WAM stack are used for many other things, the only ones we need to model at this stage are the ones that are used to represent variables in the raw goals. We consider it a strength of our approach that we can ignore the rest in our representation, since they do not affect the argument.

In our proof, we define the type for variables to have four different constructors: STACK, HEAP, ARG, and TEMP. We loosely refer to the variables that are created using the STACK constructors as being of type STACK. Initially, we only distinguish between HEAP and non-HEAP vars, then between HEAP, TEMP, and non-HEAP/non-TEMP vars, and finally between all four types of vars. However, for brevity we will initially speak as if the only kinds of vars were HEAP and STACK, then HEAP, STACK, and TEMP, where the reader should understand that STACK in the first case means non-HEAP, and in the second case means non-HEAP/non-TEMP. The meaning should be clear from context.

It should now make sense why we do not require that a stack var be of any particular type; as we progress, we will further restrict what is allowed as a stack var until only vars of type STACK are allowed, but to be able to prove theorems about stack vars at this stage and be able to continue to use them later we must give the type restriction separately.

We model heap cells using HEAP variables; we have already had to impose the constraint that HEAP vars are only bound to other HEAP vars, in constraint cphi_heap_vars of stage cphi_deriv.

Stack cells in the WAM are explicitly allocated and deallocated (by changing the top-of-stack pointer), but here we say that a STACK var is allocated if it occurs in the raw goals. One may think of the act of appending a new step to the derivation as allocating STACK vars, and the act of popping a raw goal as deallocating them, but this is only an analogy with the WAM.

We must avoid binding a STACK cell to another STACK cell that will be deallocated earlier. A stack cell may be thought of as being deallocated when the last raw goal containing it is popped; raw goals with the highest index are popped first, and within the same index from left to right. The following terms are convenient to express the constraint.

Definition 12 *The* height *of a stack var is the index into the derivation of the step where the variable occurs.*

Definition 13 *The* position *of a stack var is the index of its last occurrence in the clause body of the step where it occurs.*

By definition, a stack var must occur in some step, and by the disjointness condition this step must be unique, so the height of a stack var is well-defined.

Similarly, a stack var must occur in the body (the remaining atoms of the clause body) of the step where it occurs, so the position is well-defined.

Constraint 10 (trim_directed (1)) *If θ binds a STACK stack var v to a STACK var v', then v' must also be a stack var , and height $v \leq$ height v', and if height $v =$ height v' then position $v \leq$ position v'.*

The real effect of deallocating stack cells in the WAM is that the contents of that cell are no longer needed after that point; that is, its binding may be deleted from the store. The store is represented in derivations by the unifiers. Thus, to represent the effect of stack deallocation, we must not use these bindings after a certain point. In this stage we are adding constraints in order to make this possible.

A first (incorrect) attempt to do this is the following.

> When applying unifiers to the raw goal, only use STACK bindings that are for stack vars.

This is wrong precisely because stack vars may be reused. Say a unifier θ has a binding for a stack var v, and then more resolution steps are added to the derivation until finally the last atom containing v is popped (so v is no longer a stack var). At this point θs binding for v would no longer be used. However, at some later point v may again be used to represent a clause variable; according to this proposed constraint, the binding of θ for v would be used again!

The problem is that we stop using the binding for v once the *particular instance* of v that θ is associated with goes out of scope; it must not matter whether other instances of v come into scope. In terms of the machine, we might say that there are two different instances of a stack cell if the stack frame containing the cell were deallocated and then a new one allocated in the same place. As we shall see, the allocation of a stack frame corresponds (almost) to a resolution step.[3] Therefore if the same STACK var appears in two different resolution steps, then in the corresponding machine execution it is a stack cell that has been deallocated and then reused. Thus, we can identify different instances by their height.

Obviously, if the unifier binds a stack var , then the height of the var must be no more than the index at which the unifier occurs; subsequent instances of the var must have a greater height. The correct rule for trimming the unifiers used to compute the goal is the following.

> For each step i, if θ_i has a binding for a STACK var v, only use it if v is still a stack var and $height\ v \leq i$.

For reasons we will not go into, it is convenient to have a single predicate that describes this condition rather than a conjunction. We therefore introduce the following abbreviation.

Definition 14 (is_stack_var_Xi) $is\_stack\_var\_Xi\ bodyno\ deriv\ i\ v =$ $is\_stack\_var\ bodyno\ deriv\ v \wedge stack\_var\_height\ bodyno\ deriv\ v \leq i$

We may now give a formula that computes the next goal for this stage.

```
HD_trim_goals start x deriv =
let bodyno = deriv_bodyno deriv
and i = RES_GOAL_EL x
in
```

[3]It does not correspond exactly, because facts and chain rules do not require stack frames.

```
let body = RES_BODY (bodyno i) (EL i deriv)
and cphi = collect_phi deriv
in
HD body @|| union (compose (VAR o restrict_phi (cphi i) body)
(top_thetas' (is_stack_var_Xi bodyno deriv) start deriv))
(VAR o RES_PHI x)
```

We must now show that, with the addition of new constraints, computing the goal using trimmed unifiers gives the same result as using untrimmed unifiers, that is:

$$\mathit{HD\_trimmed\_goals}\ \mathit{gel}\ x\ \mathit{deriv} = \mathit{HD\_cphi\_deriv\_goals}\ x\ \mathit{deriv}$$

where gel is the index of the step where the raw goal occurs.

We consider a var v that occurs in the raw goal and prove that both the substitutions produce the same result. First consider the case where v is a HEAP var. Both substitutions are unions. The constraint cphi_heap_vars (which we made in stage cphi_deriv) prevents any ϕ from mapping a HEAP var, so we need only consider the first part of the union, and for the same reason we need only consider the unifiers in that part. A trimmed unifier maps a HEAP var to the same thing that an untrimmed one does, and cphi_heap_vars requires unifiers to map HEAP vars to terms that contain only HEAP vars, so the composition of any list of unifiers must produce the same thing as the composition of a trimmed list of unifiers.

Now consider the case where v is not a HEAP var. If ϕ is defined for it, then by cphi_welldefined cphi must be undefined for it, so the restriction of cphi must also be undefined for it; it remains to show that top_thetas of v is the same as top_thetas' of v, but that follows from lemma top_thetas'_stack_var0, which we prove below. Now consider the case where ϕ is undefined for v. If the cphi is undefined for v, then it maps v to a HEAP var and the argument for HEAP vars applies; otherwise lemma top_thetas'_stack_var0 again shows that top_thetas of v is the same as top_thetas' of v. $\Box$

To prove top_thetas'_stack_var0, we define a useful function and prove a general property concerning the restriction of substitutions.

Definition 15 *restrict* $P\ \theta\ v$ = *if* $P\ v$ *then* θv *else* v

Theorem 4 (restrict_P_restrict_Q_GENLIST) *For any list of substitutions* $\theta_1 \cdots \theta_n$*, predicates* $Q_1 \cdots Q_n$*, predicate* P *and variable* v*, if*

$$Pv$$
$$\forall v.Pv \rightarrow \forall i.i < n \rightarrow \forall v'.subvar\ v'(\theta_i v) \rightarrow Pv'$$
$$\forall vi.i < n \rightarrow Pv \rightarrow Q_i v$$

then $(restrict\ Q_1\theta_1 \cdots restrict\ Q_n\theta_n)v = (\theta_1 \cdots \theta_n)v$

We use induction on n. The base case is trivial. For the inductive case, we may assume

$$(restrict\ Q_1\theta_1 \cdots restrict\ Q_n\theta_n)v = (\theta_1 \cdots \theta_n)v.$$

Say v' is a subvar of $(\theta_1 \cdots \theta_n)v$. Since Pv is true by assumption, a simple inductive argument shows that Pv' is also true. This implies $Q_{n+1}v'$, so $restrict\ Q_{n+1}\theta_{n+1}v' = \theta_{n+1}v'$. $\Box$

We now prove the lemma.

Theorem 5 (top_thetas'_stack_var0) *If* v *is a stack var of* d *of height at most* i*, then for all* j *such that* $i \leq j \leq length\ d$

top_thetas' (is_stack_var_Xi (deriv_bodyno deriv) deriv) j deriv v =

top_thetas j deriv v

By applying restrict_P_restrict_Q_GENLIST, we reduce the problem to these three simpler goals.

$$Pv$$

$$\forall v.Pv \to \forall k.j \leq k \to \forall v'.subvar\ v'(\theta_k v) \to Pv'$$

$$\forall vk.j \leq k \to Pv \to is\_stack\_var\_Xi\ (deriv\_bodyno\ deriv)\ deriv\ k\ v$$

Here, Q is

is_stack_var_Xi (deriv_bodyno deriv) deriv

while the list of sigmas is the list of unifiers from j to the end of the derivation. The predicate P is free in the assumptions of the lemma, so we are free to choose whatever we want; we choose

is_stack_var_Xi (deriv_bodyno deriv) deriv j

First, we know that v is a stack var of height at most i, that is,

is_stack_var (deriv_bodyno deriv) deriv v $\wedge$

stack_var_height (deriv_bodyno deriv) deriv v $\leq$ i

This is the same thing as

is_stack_var_Xi (deriv_bodyno deriv) deriv i v

which together with $i \leq j$ implies

is_stack_var_Xi (deriv_bodyno deriv) deriv j v

which is just P v.

Second,

is_stack_var_Xi (deriv_bodyno deriv) deriv j v

implies

is_stack_var_Xi (deriv_bodyno deriv) deriv k v

since $j \leq k$, and from trim_directed and

subvar v' (θ_k v)

we may infer

is_stack_var_Xi (deriv_bodyno deriv) deriv k v'

If a var is stack var of height at most m, and $m <= n$, then obviously it is of height at most n, but the converse is of course not true in general. However, since v is bound to v', and trim_directed is true, we can show (in theorem is_stack_var_subvar, which we do not present) that

is_stack_var_Xi (deriv_bodyno deriv) deriv j v'

which is just Pv'.

Finally,

is_stack_var_Xi (deriv_bodyno deriv) deriv j v

implies

is_stack_var_Xi (deriv_bodyno deriv) deriv k v

since $j \leq k$. $\Box$.

We have now proved that when computing the goal we may trim the unifiers. In a derivation, for a raw goal that appears in step i, only the unifiers that occur in step i or greater will be used. In the WAM, unifiers are not stored separately, but rather are merged together to form the store; when dereferencing a cell, one does not take care to use only a part of the store. Therefore, we must change the way we compute the goal again by including *all* of the derivation unifiers. This will work if we ensure that the unifiers earlier in the derivation are undefined for the vars in the raw goal, that is, if

$$\forall vi.i < m \rightarrow is\_undefined\ \theta_i v$$

then

$$(\theta_1 \cdots \theta_n)v = (\theta_m \cdots \theta_n)v$$

In the case of STACK vars, this will be true because of trimming. The extra unifiers are the ones before the step where the raw goal occurs; if a STACK var occurs in this goal, then by disjointness the index of that step must be its height. If one of the extra unifiers has a binding for that var, the instance the binding is associated with must occur at its step or before, but since the raw goal occurs after it, this instance must still be on the stack, which would contradict the requirement that the vars in the raw goal must be fresh.

For HEAP vars, we just make this true by fiat.

Constraint 11 (heap_disjoint_domains) *If v is a HEAP var in the new clause, then no prior unifier may be defined for v.*

We replace this constraint in the next stage, but it is sufficient for now.

From this point on, the combination of the derivation unifiers used to compute the new goal will be called "the store". The definition will change slightly: in arg_deriv we remove ARG vars, and in fix_deriv we start taking the fixed-point of the unifiers. The important part is that they are composed separately (from the ϕs), they are trimmed, and all of them are used.

CHAPTER 18

disj_deriv: disjointness of variables

The goal list is used in classical derivations for two purposes: to compute the next goal and to determine which variables may not occur in the new clause variant. The WAM does not represent goal lists explicitly. In stage goal_deriv we stopped using goal lists to compute the next goal, and in this stage we stop using them to enforce variable disjointness.

The main difference between the disjointness condition of classical derivations and that enforced by the WAM is that unifiers play a role in the first case but none at all in the second. For example, if variable v appears in the goal list for step n, and the unifier for step $n+1$ has v in its domain but not in its range, then v cannot appear in the goal list for step $n+1$, so it may appear in the clause of step $n+2$. In a sense, it almost doesn't matter what variables appear in the clauses, since these variables can be mapped to different ones by unifiers; in contrast, the WAM store (which roughly represents derivation unifiers) plays no role at all in determining which cells may be used to represent the new clause.

In order to express the new disjointness constraint, we need a concept similar to stack var .

Definition 16 *A* heap var *of a derivation is a HEAP variable that occurs in some clause of the derivation, or is in the range of one of its ϕ mappings, or is* *contained in the original goal.*

These variables correspond to (some) heap cells in the WAM. A heap cell is allocated either to represent variables of a new clause, or is allocated during a **put** operation (which we represent using the ϕ substitution), or represents a variable in the original goal. Some heap cells represent functors and constants, of course, but we do not represent that level of detail here, and indeed never reach that level in our proof.

We also introduce a convenient shorthand term.

Definition 17 *A* deriv var *of a derivation is either a stack var of the derivation or a heap var of the derivation.*

We may now state the new disjointness condition:

Constraint 12 (disj_clause_DISJOINT) *A variable in the new clause may not be a heap var of the old derivation, may not be in the range of the new ϕ, and may not be a stack var of the old derivation after it is popped.*

We claim that this roughly models the way a WAM works. The heap cells in use by a WAM just after a **get** operation (with the exception of functors and constants) are represented by the heap vars of the corresponding derivation; the heap cells allocated by the **put** just preceding the new **get** are represented by the HEAP variables in the range of the new ϕ. Stack cells are more difficult to model because of environment trimming. After the **put** operation has processed an atom, stack cells that were in use only for that atom may in principle be immediately reused; a WAM that does this implements environment trimming. The atom that the **put** operation processes is the first raw goal of the old derivation.

Since the cells in this atom may be reused in the next clause, we must not exclude all stack vars of the old derivation, but only those that do not only occur in that atom, that is, the variables that are still in the stack after it is popped.

We are comparing this model to a simplified version of a WAM; in particular, we are ignoring the need to save stack frames for backtracking. It is an advantage of our approach that we may make this simplification; what we are establishing here is the lower limit of which stack cells may be reused; one simply cannot use stack cells for new clauses before they have stopped being used for old ones. However, later on we can strengthen this constraint to prevent the reuse of still more stack cells (which will be used to represent frames saved for backtracking). Those additional stack cells are irrelevant for the argument we make here.

The new constraint is intended to replace the old one. We must prove that if a new clause satisfies the new constraint, then it must also satisfy the old one, that is, if we were to construct the goal list for the previous step, then the new clause variables would be disjoint from the variables in that goal list. We do this by showing that the variables in the goal list are a subset of the variables excluded by our new constraint; by excluding these vars, we *a fortiori* also exclude the vars in the goal list.

First, we prove that all variables in the goal list are deriv vars. Each atom in the goal list is created by applying some unifiers and ϕ substitutions to an atom in the raw goal list. We thus want to prove that for any variable v occurring in a raw goal, only deriv vars occur in the result of applying these substitutions to v. To ensure this, we must make several restrictions on θ and ϕ. Since we want to eventually be able to model the way a WAM works, it makes sense to try to generalize the constraint that it uses.

The directedness condition in a WAM requires that heap cells be bound to heap cells (as opposed to stack cells). A WAM also requires that in some cases the heap cell being bound must have a higher address than the heap cell it is bound to, but that in fact is irrelevant for our proof. Furthermore, a WAM unification procedure does not introduce new cells; new cells are only created as part of making the new clause instance. This justifies the following constraints.

Constraint 13 (disj_theta (part 2)) *If a heap var v occurs in the domain of θ, then variables occurring in θv must also be heap vars of the new derivation.*

Constraint 14 (disj_theta (part 3)) *If θ maps a stack var v, then every variable in θv must be a deriv var of the new derivation.*

The put_unsafe_variable instruction is used to map stack cells to (fresh) heap cells if they are not bound to a heap cell or a (sufficiently low) stack cell; we model this very abstractly with the following constraint.

Constraint 15 (disj_goal_vars) *If v occurs in the goal, and v is a STACK var, then v must be a stack var of the old derivation after it is popped.*

This prevents a "dangling" stack cell from ending up in the goal. The goal is the result of applying the proper θ and ϕ substitutions to the raw goal; if a stack var that only occurred in the raw goal was unbound in all proper θs and ϕs, then it would end up in the goal, which would violate this constraint. In the next stage we will introduce new constraints that will force this to happen, but for the purpose of disjointness is is only necessary to know that it does happen, not why it happens.

Theorem 6 (wam0_goals0_vars) *For all derivations d of length n, for every v that occurs in the goals of d at step i and every v' in $(\theta_i\phi_{i+1}\theta_{i+1}\ldots\phi_n\theta_n)v$, v' is either a heap var of d or it is a stack var of d that precedes v.*

We induct on d. The base case is trivially true, since there are no steps for v to occur in if d is the empty list. For the inductive case, we assume that v occurs in the raw goals of the old derivation (which has length n) at step i, and v' in $(\theta_i\phi_{i+1}\theta_{i+1}\ldots\phi_n\theta_n)v$ is a heap var of the old derivation or a stack var that precedes v, and show that v'' in $(\phi_{n+1}\theta_{n+1})v'$ is a heap var of the new derivation or a stack var that precedes v.

If v' is a heap var of the old derivation, then cphi_phi_heap_vars prevents ϕ_{n+1} from being defined for it; v' is also a heap var of the new derivation, so disj_theta(2) forces v'' to also be a heap var of the new derivation.

Otherwise v' must be a stack var of the old derivation that precedes v. It must be the case that v' is a stack var of the old derivation after it is popped; it could only fail to be one if it only occurred in the topmost goal, in which case it would end up in $goal_{n+1}$ (since ϕ_{n+1} is undefined for v'), but then disj_goal_vars would require v' to be stack vars of the old derivation after it is popped, which would be a contradiction.

This means that v' is still a stack var of the new derivation, so the height of v and v' will be the same as they were in the old derivation, so v' will still precede v in the new derivation.

If ϕ_{n+1} is defined for v', then by cphi_phi_heap_vars the result must be a heap var of the new derivation, and by disj_theta(2) v'' must be a heap var of the new derivation. We may therefore assume that ϕ_{n+1} is undefined for v', that is, v'' occurs in $\theta_{n+1}v'$.

We may assume that θ_{n+1} is defined for v', since otherwise $v'' = v'$ and we are done. By disj_theta(3), v'' must either be a heap var or a stack var of the new derivation. If it is a stack var, trim_directed forces v'' to precede v', and therefore also v. □

Note the role that disj_goal_vars played in the proof. It requires ϕ_{n+1} to be defined for a non-HEAP var if its last occurrence on the stack is being popped and it has not been bound to a HEAP var or a stack var that is not being popped; this corresponds to the action of the put_unsafe_value instruction. Our reasoning was simplified since we did not have to specify exactly how this requirement is enforced.

We must now prove that our new disjointness constraint is at least as strong as the old constraint.

Theorem 7 (DISJOINT_g_top_phithetas) *With the appropriate preconditions, if* SNOC_PROP (disj_clause_DISJOINT g) deriv, *then* SNOC_PROP (clause_is_restricted_DISJOINT_phivars g) deriv.

If *deriv* is nil, then the goal list is just [g]; since the vars in the original goal are heap vars, they are excluded from the new clause. Otherwise, *deriv* has length $n + 1$. We must show that if v is in $goal\_list_n \phi_{n+1}$, then it cannot be in the new clause.[1] If v is in the range of ϕ_{n+1}, disj_clause_DISJOINT excludes it from the new clause. Otherwise, v occurs in $goal\_list_n$ and ϕ_{n+1} is undefined for it; by wam0_goals0_vars v must be a deriv var of the old derivation, so disj_clause_DISJOINT again excludes it from the new clause. □

[1]The old disjointness condition referred to the goal list after the last ϕ is applied.

CHAPTER 19

put_deriv: unsafe stack variables

In this section we give conditions that more closely correspond to the way the WAM performs delayed heap allocation.

Delayed heap allocation happens in two very different ways in a WAM. Both concern variables that only occur in the clause body. In one case, the machine represents the variable by a stack cell and "hopes" that it becomes bound to a heap cell (or a lower stack cell) by the time the last atom it occurs in is processed; if not, it allocates a heap cell for it. In the other case it just immediately allocates a heap cell for the variable when the first atom containing it is processed.

In prior stages we required that all stack vars in the goal must still be in scope after the stack is popped, but did not make any assumptions concerning how that was guaranteed. We will now add such assumptions, dividing the problem into two cases, but will assume nothing about which technique is used for a particular case. This is another example of delaying the introduction of implementation decisions into the proof until they are actually needed.

The main constraints this stage eliminates are disj_goal_vars, disj_theta, and cphi_welldefined, and cphi_thetas_undefined.

19.0.1 TEMP body vars

We introduce the class of TEMP vars for those body variables that we will not represent using stack cells; what we referred to as STACK vars in the previous stage were actually both STACK and TEMP vars. This gives us three categories: HEAP, STACK and TEMP. We impose the following condition on TEMP vars:

Constraint 16 (put_temp) *For TEMP var v, the current ϕ is defined for v iff v is in the raw goal and no atom prior to the raw goal in that clause contains it.*

This models what a WAM does; it allocates a new heap cell for such variables as soon as it encounters the first atom containing it (right-to-left direction of iff), and it does *not* allocate a heap cell for the variable again when processing a subsequent atom (left-to-right direction of iff).

This new condition allows us to drop cphi_welldefined (constraint 5) for TEMP vars;[1] if ϕ is defined for a TEMP var v, put_temp prohibits v from occurring in an earlier goal, while on the other hand if an earlier ϕ were defined for v then then cphi_phi_domain would require v to occur in an earlier goal, so cphi must be undefined for v.

We show that disj_goal_vars (constraint 15) is satisfied by showing that TEMP vars simply cannot occur in the raw goal. To do this, we add constraints that prevent TEMP vars from ever occurring in the domain or range of a unifier. This is certainly desirable; TEMP vars are only a notational convenience and we eventually want to eliminate them.

[1] Constraint cphi_welldefined was stated in terms of STACK vars, but we have now "split" that category into STACK and TEMP vars, so TEMP vars in this stage must satisfy constraints we still placed on STACK vars in the previous stage.

Consider what kinds of bindings a unification routine makes; given two terms, it binds variables that occur in one or the other of the terms to subparts of the other; no one writes unification routines that bind existing variables to terms containing new variables.[2] We may therefore impose the following constraint:

Constraint 17 (theta_goal_head) *If a variable occurs in the domain of θ, then it must occur either in the clause head or the goal.*
If a variable occurs in the range of θ, then it must occur in the goal or it must occur in the clause head and be a HEAP var.

The following condition is reasonable since we use TEMP vars as placeholders for variables that occur only in the clause body.

Constraint 18 (put_temp_conds) *The clause head may not contain TEMP variables.*

The following predicate is handy to state our theorem.

Definition 18 *no_temps_in_theta is true of a step iff no TEMP var occurs in the domain or range of the θ for that step.*

These two conditions allow us to prove the following.

Theorem 8 (no_temps_in_theta)

SNOC_PROP theta_goal_head d $\wedge$ ALL_EL put_temps_conds d $\rightarrow$

ALL_EL no_temps_in_theta d

[2] Here we are discussing unification routines. The process of "unifying" a head with a goal in the WAM actually simultaneously creates a clause variant and a unifier; although the processes are interleaved, one may think of them as being consecutive.

We use induction on d. The base case is trivial, but we must consider the case of a derivation with just one step. The initial goal contains only HEAP vars and the clause head may not contain TEMP vars, so theta_goal_head prevents the initial unifier from containing TEMP vars.

For the inductive case, we first show that no TEMP var may occur in the goal. Say some TEMP var v did occur in *HD_put_goals x deriv v*. It can't be because ϕ introduced it, since ϕs may only have HEAP vars in their range; by inductive hypothesis the store cannot have introduced it; and the earlier relevant ϕs cannot have introduced it either, so it would have to have occurred in the raw goal. But then either this would have been the first occurrence in the body, in which case put_temp would have forced ϕ to map v, or it would be the second or subsequent occurrence in the body, in which case some prior relevant ϕ would have been forced to map it, so it couldn't appear in the goal after all.

Since the goal may not have TEMP vars in it, and put_temp_conds prevents the clause head from having TEMP vars in it, theta_goal_head prevents θ from having TEMP vars in its domain or range. $\Box$

We may drop disj_theta for TEMP vars since no unifier is ever defined for TEMP vars. It is also easy to show that phi_thetas_undefined may be dropped for TEMP vars, since the composition of unifiers must also be undefined for all TEMP vars, and if ϕ is defined for a TEMP var v then put_temp forces all earlier relevant ϕs to be undefined for v.

19.0.2 STACK body vars

The much more difficult case is that of STACK vars. First we define a predicate to describe the conditions under which a STACK var must be allocated a HEAP var.

Definition 19 *A stack var is* unsafe *if*

- *it is of type STACK*
- *it is in the raw goal*
- *it does not occur after the raw goal in the clause body*
- *it does not occur in the head*

A variable that we call unsafe does not correspond to a stack cell in the WAM that is called unsafe; this is really just a convenient predicate for identifying those instances of STACK vars that may be *potentially* unsafe.

When describing these constraints, we shall use the term *store* to mean the composition of all the trimmed unifiers in the derivation (which is what we started using to compute the goal in stage trim_deriv). The first constraint we add corresponds to when the last atom containing the stack cell is processed and it is determined that the cell is unbound; in that case, a new heap cell must be allocated.

Constraint 19 (unsafe_stack (part 1)) *If v is unsafe, and the store is undefined for v, then ϕ must be defined for v.*

The next constraint corresponds to the situation where the stack cell is bound to another stack cell, but the other cell is not "low enough", so both get bound to a new heap cell. The only time both the store and ϕ may be defined for an (unsafe) variable v is when v dereferences to another unsafe stack var.

Constraint 20 (unsafe_stack (part 2)) *If v is unsafe, and the store is defined for v, and ϕ maps v to hv, then the store must map v to a STACK variable (call it v'), and $\phi v' = \phi v$.*

The third constraint says that ϕ only maps stack cells that are unsafe.

Constraint 21 (unsafe_stack (part 3)) *If ϕ is defined for v, and v is a STACK variable, then v must be unsafe.*

The last constraint is difficult to understand from the perspective of the WAM.

Constraint 22 (unsafe_stack (part 4)) *If v is unsafe, and the store is defined for v, and v' is a subvar of* store v, *and v' is a STACK variable, and ϕ is defined for v', then ϕ must also be defined for v.*

Say stack cell c is bound to stack cell c' and c' is not low enough; then the **put** operation will allocate a new heap cell h and bind c' to it. This means that whenever c or c' is encountered in the atom being processed, the cell h will be put instead of c'. We represent this effect by saying that the corresponding ϕ maps both c and c' to h. Obviously, a WAM cannot bind c' to h and not have c dereference to h as well; that is all that this constraint expresses.

The earlier constraint cphi_welldefined, which we have shown above to be satisfied by the new constraints for TEMP vars, is also satisfied for STACK vars for a similar reason. As before, each ϕ is associated with a unique atom in a clause body, but in the case of stack vars it is the atom containing the last, not the first, occurrence of the variable; the argument is the same, though.

The proof that these new constraints (in addition to others we do not show here) satisfy cphi_thetas_undefined is very difficult and will not be described here.

19.0.3 Collecting ϕs

At the same time that we refine the conditions for ϕ into TEMP and STACK cases, we also refine the way we collect prior relevant ϕs. Until now, when computing the next goal, for every variable in the raw goal we checked all prior relevant ϕs to see if they had a binding for it. If the WAM behaved like this, the put_value instruction would have to check if a heap cell had been allocated for the cell being processed; of course, nothing like this happens. Since we now require that STACK vars only be mapped when their last occurrence is processed, it must be the case that all prior relevant ϕs are undefined. This means there is no need to check whether prior ϕs are defined for them. Since we now require TEMP vars to be mapped when their first occurrence is processed, we know whether they have been mapped simply by checking the position of the atom being processed. Again, we do not need to check if a prior ϕ is bound for them, but we must (almost) always save the heap cells allocated.[3]

19.0.4 Eliminating disj_theta

The constraint disj_theta is replaced by theta_goal_head in this stage, because disj_theta uses the predicates stack var and heap var, while theta_goal_head does not. Early on, the author attempted to use a constraint like theta_goal_head rather than disj_theta, but was unable to prove the necessary proofs; it was necessary to know that the domain and range of θ contained (roughly) only deriv vars. He carried disj_theta (and several other predicates that use heap var and stack var) through many subsequent stages, that is, every time the concept of derivation changed, he created new predicates that corresponded to heap var and stack

[3]The exception is where a variable occurs in only one body atom.

var in the new system and defined new constraints that used the new versions of heap var and stack var. It gradually became clear that either these constraints would have to be eliminated (if they were consequences of other constraints placed on the derivations) or one would have to prove that the final unification routine satisfies them. It seemed likely that they were in fact consequences of the other constraints; it also seemed likely that it would be difficult to prove something like disj_theta in a late stage, whereas a constraint like theta_goal_head is easy to prove.

The only constraint eliminated in this stage that uses heap var or stack var is disj_theta. The following definitions are convenient for proving that other constraints imply disj_theta.

Definition 20 *v is a* deriv_var_aux *of non-nil derivation d if it is a deriv var of d, but it does not occur in the last clause of d.*

Definition 21 *goal_deriv_vars x deriv is true iff every var v in the goal of step x is a deriv_var_aux of (SNOC x deriv)*

The first thing we prove is that the goal at each step contains only deriv_var_aux vars (theorem goal_deriv_var_aux). In this proof we assume that disj_theta is true for all derivation steps but the last; it is very roughly of the form:

$$\textit{SNOC\_PROP disj\_theta (BUTLAST d)} \wedge \textit{ALL\_EL theta\_goal\_head d} \rightarrow$$
$$\textit{SNOC\_PROP goal\_deriv\_vars d}$$

We prove separately a theorem called put_theta_lemma which is very roughly of the form:

$$\textit{SNOC\_PROP goal\_deriv\_vars d} \rightarrow \textit{SNOC\_PROP disj\_theta d}$$

We wish to prove a theorem very roughly of the form:

ALL_EL theta_goal_head d $\rightarrow$ SNOC_PROP disj_theta d

We do this by first strengthening the proposition to this.

ALL_EL theta_goal_head d $\rightarrow$

SNOC_PROP goal_deriv_vars d $\wedge$ SNOC_PROP disj_theta d

The first inductive subgoal is:

ALL_EL theta_goal_head d $\wedge$ theta_goal_head x $\wedge$

SNOC_PROP disj_theta d $\wedge$

SNOC_PROP goal_deriv_vars d $\rightarrow$

SNOC_PROP goal_deriv_vars (SNOC x d)

We prove this by applying goal_deriv_var_aux (where we instantiate the d in that theorem to *SNOC* x d and expand ALL_EL).

The second subgoal is:

ALL_EL theta_goal_head d $\wedge$ theta_goal_head x $\wedge$

SNOC_PROP disj_theta d $\wedge$

SNOC_PROP goal_deriv_vars d $\wedge$

SNOC_PROP goal_deriv_vars (SNOC x d) $\rightarrow$

SNOC_PROP disj_theta (SNOC x d)

Note that we may use the result of the first subgoal to prove the second subgoal; we prove it by applying put_theta_lemma (where again we instantiate d with SNOC x d). The reader should understand that the conjuncts must be ordered this way in order to be able to apply theorems goal_deriv_var_aux and put_theta_lemma. In the actual theorem (which is called theta_lemma) there are actually 14 conjuncts in the conclusion (rather than just two as in this simplified sketch); finding an ordering that would work and paring down the assumptions for each of the subproofs was not easy.

Elsewhere, the author argues that an advantage of this approach over the so-called machine-oriented approach is its simplicity. In this case, although the concepts involved still seem simple, the way that they must be organized in order to complete the proof is in fact difficult; this makes it less clear that it is in fact simpler than the alternative.

CHAPTER 20

arg_deriv: adding argument registers

By this stage we have divided variables into three categories: HEAP, STACK and TEMP vars. The fourth (and final) category is ARG vars, which correspond to WAM argument registers. An argument register may be used by the WAM in one of three different ways: it may represent a variable (just after **get** has completed), it may hold part of the goal (just after **put** has completed), or it may simply temporarily hold the address of a newly allocated heap cell (during the execution of either **get** or **put**). We treat the first use in this section, the second in a later section, and never reach the stage appropriate for dealing with the last use. We believe that it is an advantage of our proof that it deals with these uses separately, making the operation of the WAM clearer.

We now split the category of STACK vars again into STACK and ARG vars. To avoid confusion, what we used to refer to as STACK vars we will now refer to as ARG/STACK vars.

In addition to modeling argument registers, we introduce top-of-stack and top-of-heap counters to eliminate the predicates stack var and heap var, and thus to eliminate the constraints that use these predicates; the main ones are stack_disjoint, put_stack_directed, and disj_clause_DISJOINT. There is no particular reason why this should be done at this stage; it just seemed easier to add this to this stage than to insert a new one.

20.0.5 Computing the goal

We start by adding a constraint on where ARG vars may appear in a clause.

Constraint 23 (arg_in_head_not_in_TL_body) *If an ARG var appears in a clause, then it must appear in the head and it may not appear in the tail of the body.*

This agrees with the WAM. As we have already noted, this is not intended to cover all the cases where the WAM uses an argument register, but rather only the case where one is actually bound to an input cell during unification (when the WAM is in read mode). Only in this case may the register be thought of as representing a variable in the clause variant; our approach has the advantage of making this distinction very clear.

This constraint implies that the only place an ARG var may appear in the stack is in the clause body of the last derivation step; this is because every other body in the derivation must have been popped at least once (if it is non-nil), so only the second and subsequent atoms in the body remain, and constraint arg_in_head_not_in_TL_body prevents variables in these atoms from being ARG vars.

We make restrictions on where ARG vars may appear in order to specialize the way the goal is computed. We have shown in stage trim_deriv that we can ignore unifier bindings for ARG/STACK vars that are not on the stack when computing the goal. Instead of trimming the bindings of ARG vars, we will accomplish the same thing by dividing the unifier into two parts: one that contains only ARG vars in its domain (which we will call REGS), and another that contains no ARG vars in its domain (which we will call θ).[1] The unifier is now created by composing REGS with θ (we explain in stage fix_deriv why we do not take the union).

The fundamental constraint we impose on REGS is the following.

Constraint 24 (arg_vars_in_args) *Only ARG vars may occur in the domain of REGS, and no ARG vars may occur in the range of REGS.*

This makes sense from the point of view of the WAM; the REGS substitution represents (only) what the WAM argument registers point to, and they never point to other argument registers.

Related to this is the restriction on θ concerning ARG variables.

Constraint 25 (no_arg_vars_in_theta) *ARG vars may not occur either in the domain or the range of θ.*

The θ substitution is intended to represent the contents of the WAM store, that is, some of its heap and stack cells; therefore argument registers cannot be in its domain, and they can't be in its range either, since WAM cells are never interpreted as pointing to argument registers.

As we have seen, the only atom in the stack that can have ARG vars in it is the the first atom in the last clause body of the derivation. To compute the goal, we will apply the REGS substitution to this atom. We have already excluded ARG vars from the range of REGS and θ, so to prevent ARG vars from ending up in the goal we need only require that REGS map all the vars in that atom.

Constraint 26 (theta_is_restricted2_arg_domain') *REGS must be defined for all ARG vars that occur in the clause.*

[1] Note that θ has referred to the unifier until this point.

This makes sense; the argument registers that we are modeling in this stage all appear in the head, and they always get bound to something during unification.

Consider how the process of computing the goal is affected by these constraints. A complex substitution was applied to the raw goal to obtain the goal, one element of which was the "store", which at this point is just the composition of all the trimmed unifiers in the derivation. Now that we have split the unifier into two parts, it is the composition of the trimmed compositions of REGS and θ substitutions. We now prove that one can drop all REGS substitutions but the first.

Theorem 9 *$(REGS_i \circ \theta_i \circ \ldots REGS_n \circ \theta_n)v = (REGS_i \circ \theta_i \circ \ldots \circ \theta_n)v$, and no ARG vars occur in either of these terms, where i is the step where the raw goal appears, n is the length of the derivation ($i <= n$), and v is an ARG var that occurs in the raw goal.*

Use induction on n. If $i = n$, then the two terms are obviously identical; constraint theta_is_restricted2_arg_domain' forces $REGS_i$ to be defined for v, arg_vars_in_args prevents ARG vars from appearing in the result, and no_arg_vars_in_theta prevents θ_i from introducing ARG vars.

For the inductive case, assuming the statement is true for n we have $((REGS_i \circ \theta_i \circ REGS_{i+1} \circ \theta_{i+1} \circ \ldots REGS_n \circ \theta_n) \circ (REGS_{n+1} \circ \theta_{n+1}))v$
$= ((REGS_i \circ \theta_i \circ \theta_{i+1} \circ \ldots \circ \theta_n) \circ (REGS_{n+1} \circ \theta_{n+1}))v$
$= (((REGS_i \circ \theta_i \circ \theta_{i+1} \circ \ldots \circ \theta_n)v)REGS_{n+1})\theta_{n+1}$
$= ((REGS_i \circ \theta_i \circ \theta_{i+1} \circ \ldots \circ \theta_n)v)\theta_{n+1}$
$= (REGS_i \circ \theta_i \circ \theta_{i+1} \circ \ldots \circ \theta_n \circ \theta_{n+1})v$ since by inductive hypothesis the term $REGS_{n+1}$ is applied to contains no ARG vars, and by arg_vars_in_args it can

only be defined for ARG vars. Again, no_arg_vars_in_theta prevents θ_{n+1} from introducing ARG vars. □

Pictorially we will have the following situation:

$$
\begin{array}{lcl}
(REGS_i \circ \theta_i \circ & & (REGS_i \circ \theta_i \circ \\
REGS_{i+1} \circ \theta_{i+1} \circ & & \theta_{i+1} \\
\cdots & = & \cdots \\
REGS_n \circ \theta_n)\ \mathrm{v} & & \theta_n)\ \mathrm{v}
\end{array}
$$

Theorem 9 tells us we need only use the REGS substitution of the step the raw goal occurs in; in fact, sometimes we don't even need that one. We have seen above that if an ARG var occurs in the stack, it must be in the last derivation step; therefore, if the raw goal isn't in the last step, that is, if the clause in the last step has no body, then we need not use any REGS substitution at all. We can decide whether we need to use the REGS substitution by checking the bodyno of the raw goal; the bodyno for a step is zero iff it is the last.

For clarity we will explicitly review what the analogous situation in the machine is. The question is whether or not to use the contents of the argument registers just after the `get` for matching a clause head have all succeeded. If the body of the clause just matched is non-empty, then the answer is yes; it may need the argument registers to `put` the first body atom. Otherwise, the answer is no; the machine will execute a `proceed` instruction and `put` the next atom in the caller, in which case the contents of the argument registers are irrelevant.

We now begin to use a more sophisticated system to compute the substitution for the goal; because it is more complex, we give the substitution separately. The new composite substitution is:

if v is a TEMP

then if it occurs earlier in the clause body

    then store (cphi v)

    else ϕv

else if this is the first atom in the clause body

    then if v is an ARG

        then (ARGS v) store

        else store v

else if v is a STACK

    then if ϕ is undefined for v

        then store v

        else ϕv

else store v

where cphi is the collected relevant ϕs, store is the composition of the trimmed θs, and body is the clause body of the step where the raw goal appears.

The formula we now use to compute the goal to be unified is now:

HD (RES_BODY (bodyno i) (EL i deriv))

    arg_goal_subst OF WHAT

This method computes the same goal as the old one, given our new constraints, that is,

HD_arg_goals x deriv v $=$ *HD_trim_goals 0 x deriv v*

We will consider all the cases, where we may assume that v occurs in the raw goal. Theorem 9 allows us to equate the expression computing *store* in the two expressions. Consider the case where v is a TEMP. If it has already occurred in the clause body, the left side returns *store (cphi v)*; since some prior relevant ϕ must be defined for it (by put_temp), ϕ must be undefined for v, so the right side simplifies to the same expression. If this is the first occurrence of v, then the left side returns just ϕv; put_temp requires ϕ to be defined for v, so the prior relevant ϕs must be undefined for v, and the unifiers are undefined for it since it is a TEMP, so the left side simplifies to the same thing. Consider the case where *bodyno* $i = 0$, that is, the raw goal comes from the last step. If v is an ARG, the left side returns *(args v) store*; since ϕ and all relevant ϕs must be undefined for v (by no_arg_vars_in_phi), so the right side simplifies to the same thing. If v is not an ARG, the left side just returns *store* v; there can be no relevant ϕs, and since v cannot be a TEMP (we have already considered that case) ϕ cannot be defined for v (since it could only be defined if v were a STACK var, in which case it would have to be v's last occurrence, which cannot be since this is the first atom in the body and STACK vars cannot appear only there), so the right side simplifies to the same thing. Consider the case where v is a STACK var. If ϕ is undefined for v, the left side returns *store* v; no prior relevant ϕ will ever be defined for a STACK var that appears in the raw goal (since put_unsafe requires that only the last occurrence be put), so the right hand side is the same. If ϕ is defined for v, the left side returns ϕv; again, no prior relevant ϕ will ever be defined for the STACK var, and even though the store may be defined for v, our definition of union gives the second substitution priority over the first, so the right side computes the same thing. If none of the three preceding cases is true, then the left side returns *store* v; since both ϕ and all prior relevant ϕs must be undefined (since v cannot be either a TEMP or STACK var), the right side simplifies to the same thing. $\Box$

The new formulation just does some case analysis and simplification; it is desirable because it more closely resembles the way a WAM works. We will go through each of the cases and make this clear. Case one: v is a TEMP. This represents the case where v is a body variable and the compiler has chosen (for whatever reason) to not use a stack cell to represent it. If the compiler is processing a body atom and encounters such a variable that it has not seen before, then it will emit a put_variable instruction, which will result in a heap cell being allocated (which is represented by a ϕ binding). If this variable occurs in subsequent atoms, then the compiler will also reserve a stack slot and save the new heap cell there; this is represented by the cphi substitution, which is just the union of the prior relevant ϕs. Otherwise, the compiler saves it in a argument register, but this level of detail is not represented here. If the variable has been seen before, then the compiler will simply put the heap cell that is recorded in the stack slot that was reserved for this variable when it was first encountered; that is represented by the expression *cphi* v. When computing the goal, we must also apply *store* to that; the goal in the WAM is not just the contents of the argument registers after the **put**, but rather the *dereferenced* contents of these registers.

Otherwise, this may be the first body atom. If the compiler has represented the variable using an argument register, that is, if it used this register with a get_variable instruction when processing the clause head, then it will emit a put_value instruction, which will load the contents of this register into another register; the contents is represented by $ARGS\ v$. Otherwise, a put_value instruction (that does not use an argument register) will be emitted; in this case, a heap cell would have been allocated to represent the variable, which would just be v

in our system.

Otherwise the compiler may have chosen to represent the variable using a stack cell (which would correspond to v). Our expression does not take all cases into account; it should distinguish between non-final occurrences (in which case ϕ is never used) and final ones (in which case it may be used). In a non-final occurrence, the compiler would emit a put_value instruction; since ϕ must be undefined in this case, our expression corresponds. In the first occurrence in the final atom in which it occurs, the compiler would emit a put_unsafe_variable instruction. The case split in our expression corresponds purely to runtime behavior; if the instruction determines that it must allocate a new heap cell, then the corresponding ϕ will be defined, and the result is just the heap cell, otherwise it is just the result of dereferencing the stack cell. Subsequent occurrences would use put_value, but would effectively be doing the same thing.

Finally, if none of the other cases apply, the compiler will just emit a put_value instruction.

This is still not quite what the WAM does. Here, we distinguish neatly between values that we think of as being loaded into the output registers (such as v or *cphi* v) and the dereferenced value used for the goal (*store* v or *store (cphi* v*)*). The put instructions may not do that; they may or may not load a dereferenced value into their arguments. The correspondence is not exact, but it is fairly close.

The other difference is that we do not represent the building of structures here. A structure is simply part of the atom being put; we apply substitutions to its variables but do not need to create the structures. At a later stage (which we do not reach) one would add these details.

20.0.6 Easy refinements of constraints

In stage cphi_deriv, the constraint cphi_phi_range allows ϕ to be defined for v only if the unifier at the step where the goal comes from is undefined for v. We have now required REGS to be defined for all ARG vars, and since θ cannot have ARG vars in its domain the unifier (which is $REGS \circ \theta$) cannot be undefined for any ARG var. Therefore, we may satisfy that earlier constraint by imposing the following one.

Constraint 27 (no_arg_vars_in_phi) *ϕ may not be defined for an ARG var.*

Similarly, by expanding the old constraint theta_goal_head and specializing it for ARG vars we get the following.

Constraint 28 (arg_goal_head) *If REGS is defined for v, then v must occur in the clause head.*
If a variable occurs in the range of REGS, then it must either occur in the goal or it must be a HEAP var and occur in the clause head.

Note that in the first condition we no longer allow the case where the variable occurs in the goal; that is because we can prove that ARG vars never occur in the goal.

We must make a trivial change to the constraint that the unifier be a mgu.

Constraint 29 (arg_mgu_ok) *The substitution $REGS \circ \theta$ must be a mgu for the clause head and the goal.*

20.0.7 "Pulling" properties from a previous stage

Until now, we have (mostly) been concerned with defining increasingly specialized properties to replace more abstract ones; in this stage, we also want our more specialized properties to *inherit* the characteristics of prior ones. That is, if in stage i we have proved that $D_i d \rightarrow P d$ (where D_i is the derivation predicate for stage i), then we may want to know that $D_{i+1} d \rightarrow P' d$, where P' is a slightly different formulation of P. The simplest example concerns TEMP vars in unifiers. We proved in the previous refinement that no TEMP vars may appear in the unifier (no_temps_in_theta). In this stage, we prove the closely related property that no TEMP vars may appear in θ as follows. We map an arg_deriv derivation to a put_deriv derivation by composing REGS and θ of the arg_deriv to form the unifier of the put_deriv and dropping the REGS substitution. Theorem no_temps_in_theta says that a put_deriv unifier may not have TEMP vars in it; therefore, no TEMP vars may appear in $ARGS \circ \theta$. But this means that no TEMP var may occur in θ, since if one occurs in the domain (or range) of θ, then it must also occur in the domain (or range) of $ARGS \circ \theta$. Therefore no TEMP vars may occur in the θs of an arg_deriv deriv.

The general flow of such proofs is shown in figure 20.1. That is, to show that the refined hypothesis implies the refined conclusion, rather than carry out the proof again using the concepts in the refined stage, we first translate the hypothesis into the abstract form using our mapping, apply the existing theorem to get the abstract conclusion, and show that if that conclusion holds, then the refined conclusion must also hold. The only new part that must be provided is the last part; mapping the hypotheses is nothing more than applying the mapping to them, and applying the theorem is trivial.

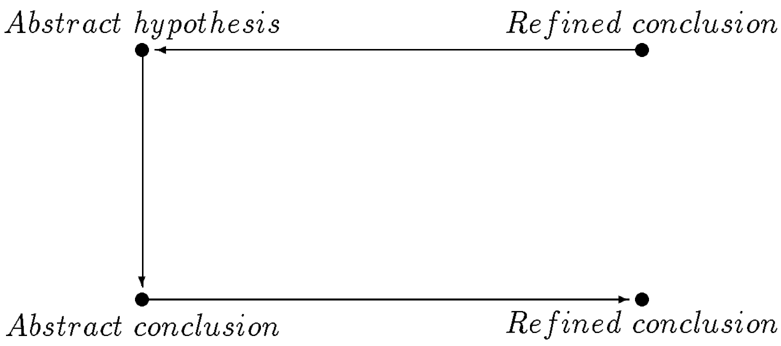

Figure 20.1: Transforming an abstract property into a refined one.

20.0.8 Eliminating heap var and stack var

Although there is no logical reason for doing so, in this stack we introduce top-of-stack and top-of-heap values in order to eliminate the predicates heap var and stack var; these values eventually become the WAM `E` and `H` registers. The constraints we eliminate concern disjointness and directedness. The new predicates we introduce are not identical to the old ones, in that they are true for more variables than the old ones, but they are good enough to replace them in these two cases. The predicate stack var is still used to compute the trimmed θs; for no particular reason, that is not eliminated until the next stage (fix_deriv).

Constraint 30 (top_stack_ok) *If the new clause body has more than one atom, then the address of every STACK var in the clause must be at least as large as TOP_STACK.*

If the new clause body has more than one atom, and the stack of the old derivation after it has been popped is not empty, then the new TOP_STACK must be larger than the addresses of all variables that occur in any remaining body atom on the top of the stack (after it has been popped), and it must be at least as large as the TOP_STACK value for the resolution step where these atoms *occur.*

This constraint is complex, and it was formulated this way only after a subsequent stage had been started and it was discovered that an earlier, simpler formulation was inadequate. We are careful not to require anything of a new clause that has fewer than two body atoms, because in such cases the WAM will not change the stack pointer. The TOP_STACK of the new step divides the addresses of the stack vars of the new derivation into two parts: those that occur in the new clause (which must at least as large), and those that do not (which must be smaller). The first property holds because we require it in the constraint, but the second property holds because we can prove that it is a consequence of the constraint. The second constraint can be understood as follows. After the WAM processes a raw goal, it sets the stack pointer for the call. If this is the last goal in the stack frame, and the stack is empty after the frame is popped, then (for our purposes) the stack pointer may be anything. If there are no goals left in the stack frame, then the WAM will use the saved stack pointer in the frame just below (which corresponds to the TOPSTACK value of the step in which that goal occurs). Should there still be goals left in the stack frame, then it will pass a top stack value that is at least as large as the saved stack pointer in the frame below, and is also at least as large as the remaining goals in this stack frame.

We must use two values as limits for HEAP vars, because there are two points at which we allocate heap vars in each resolution step. The value TOPPHIHEAP is the lower limit for new vars allocated by ϕ, and TOPHEAP is the lower limit for new vars allocated for the new clause. Eventually, these merge into the `H` register.

Constraint 31 (top_heap_ok) *The initial TOPPHIHEAP is greater than all HEAP cells in the original goal.*

All addresses of HEAP vars in the range of ϕ must be between the TOPHEAP of the previous step and the TOPPHIHEAP of the current step.

TOPHEAP must be at least as large as TOPPHIHEAP.

The addresses of all HEAP vars in the new clause must be between TOPPHIHEAP and TOPHEAP.

TOPPHIHEAP must be at least as large as the TOPHEAP of the last resolution step.

This limits heap addresses for each step in the following way:

topheap

    clause vars

topphiheap

    phi vars (or original goal)

Since we now have specialized disjointness conditions for STACK and HEAP vars, we need a new condition for TEMP vars. In a subsequent stage, we will see that this can be eliminated.

Constraint 32 (arg_temps_ok) *All TEMP vars in the new clause are distinct from TEMP vars used in all prior clauses.*

One of the constraints we eliminate is the disjointness constraint; this says that existing heap vars and stack vars may not be used for the new clause. We will prove that we can bound the set of heap vars and stack vars using TOP_STACK, TOPPHIHEAP and TOPHEAP; we cannot exactly determine the set, but we can use these values to guarantee that disjointness holds. Thus, we will replace the condition "v is not a heap var " by "v is greater than TOPHEAP", which is corresponds more closely to the WAM.

Theorem 10 (is_heap_var_less_topheap) *If v is a heap var, then the address of v is less than TOPHEAP.*

We use induction on the derivation. No vars are heap vars of an empty derivation. In the inductive case, a heap var of the new derivation may be a heap var of the old derivation, or a subvar of the new clause, or in the range of ϕ. If it is a heap var of the old derivation, then by inductive hypothesis its address must be less than the TOPHEAP of the next-to-last step. Constraint top_heap_ok requires this to be at most TOPPHIHEAP of the last step, which in turn may be at most TOPHEAP of the last step. If v is in the range of ϕ, its address must be less than TOPPHIHEAP, which must be at most TOPHEAP. If v is in the new clause, then its address must be less than TOPHEAP. □

Proving a similar theorem for STACK vars is harder.

Theorem 11 (top_stack_ok_prop) *If v is a stack var that occurs at some step less than i, and the clause at step i has at least two atoms, then the address of v is less than the TOPSTACK of step i.*

This just states the idea that the TOPSTACK of a step separates the stack vars at that step and higher from the ones before that step; we require that the clause have at least two atoms because otherwise we do not want to have to use the TOPSTACK.[2] We omit the proof.

[2]Think of how the WAM stack pointer does not change when such clauses are called.

The other constraint using stack var that we eliminate is the directedness condition (put_stack_directed). This was phrased using the height and position of stack vars. The constraint top_stack_ok already enforces an ordering among variables in different steps of the derivation; all we need to do is add an ordering for STACK vars in a given step.

Constraint 33 (clause_directed) *If STACK vars v and v' occurs in the clause body, and v's last occurrence is after vs, then the address of v' must be less than the address of v.*

This ensures that stack cells that are freed later have lower addresses.

CHAPTER 21

fix_deriv: modeling dereferencing

Thus far our model of the store has been the composition of trimmed θs. The WAM does nothing like this; if the WAM store is thought of as encoding a substitution, then the act of dereferencing the store corresponds to applying the substitution repeatedly until nothing changes. This corresponds to taking the fixed-point of the substitution. This refinement shows that, given some extra conditions, we may use the fixed-point of the union of the θs rather than their composition in order to compute the goal.

Not all substitutions have a sensible fixed-point.

Definition 22 *A substitution is* directed *if there is a function f from variables to natural numbers such that whenever the substitution maps v to a term containing v', then $fv' < fv$, and there is an n such that $fv < n$ for all v.*

The n provides a bound; after applying the substitution n times, the substitution can no longer change a term. We need a bound because we allow infinite substitutions.

This is a general condition for determining when it makes sense to talk about the fixed-point of a substitution. We may satisfy this condition for a finite substitution by showing that it may be decomposed into a composition of substitutions, each of which has only one variable in its domain, where the range of each substitution does not contain variables that appear in the domain of substitutions earlier in the list. We could then create the required function f by assigning to each variable the position in the list where the substitution containing it in its domain appears; the bound n would be the length of the list. Eventually, the bindings generated by the WAM unification routines will have to be shown to be directed; all we need do is show that these routines generate bindings sequentially like this. This gives us confidence that we will eventually be able to satisfy the following constraint.

Constraint 34 (theta_is_directed) *θ in directed.*

New θs are not kept separate from old ones, but are combined with the store using a union operation. Thus, we also need a sufficient condition (that the WAM can easily fulfill) that guarantees that the union of two (directed) substitutions is directed.

Definition 23 σ precedes θ *if no variable in the domain of σ appears in the range of θ.*

What this means is that σ can map into the domain of θ, but not vice versa. We must prove that the union of two directed substitutions is directed, because the store will be built by taking the union of substitutions.

Theorem 12 (subst_is_directed_union_cond) *If σ and θ are directed, and σ precedes θ, and they have disjoint domains, then $\sigma \cup \theta$ is directed.*

In addition to showing that the union is directed, we prove that the fixed-point of the union is equal to the compose of the fixed-points, because we must show that when we take the fixed-point of the new store (which is the union of all the θs), we get the same thing as the old store (which is the composition of the fixed-points of all the θs).

Theorem 13 (subst_fix_union) *If σ and θ are directed, and σ precedes θ, and they have disjoint domains, then $fix(\sigma \cup \theta) = fix(\sigma) \circ fix(\theta)$*

We now require:

Constraint 35 (theta_is_restricted2_subst_precede1) *The store must precede the new unifier.*

Constraint 36 (theta_is_restricted2_domains_disjoint) *The domains of θ and the store are disjoint.*

This is exactly what the WAM does. If the WAM unification process binds cell x to cell y (which means that y appears in the range of the unifier), then y must have originally been unbound (which means that y cannot have occurred in the domain of the corresponding abstract store); if a binding had already existed, then then the unification process would have dereferenced that binding. Furthermore, if a unification routine binds a cell, then that cell must have been unbound in the store (almost).

We say "almost" because some unification routines also generate bindings during the dereferencing process; each cell in the dereferencing chain is rebound to the final result so that subsequent dereference operations require fewer steps. This appears to contradict this requirement, but it does not. *These bindings are not part of the unifier;* it is not just that they need not be part of the unifier, they *cannot* be. If one wanted to model this process, one would add another substitution to represent the additional bindings and show that adding this substitution to the rest of the store produces the same goal.

In most cases we do not change the constraints imposed in stage arg_deriv. The mapping from a fix_deriv derivation to an arg_deriv derivation simply replaces θ with fix(θ), so if a constraint does not refer to θ then it is trivial to show that it is preserved. Even in cases where θ is used it is often not hard.

Nearly all uses of the predicates stack_var and heap_var were eliminated in arg_deriv; this is desirable because constraints using these predicates would probably be difficult to satisfy in the final implementation. The one remaining constraint concerns the computation of the goal; in that constraint the θs were still trimmed using the predicate stack_var. This stage eliminates this use by trimming the θs a different way. The new way is recursive and similar to the way the WAM works. Roughly speaking, as each new step is added and we pop a raw goal from the stack, we also trim bindings from all of the old θs that are for STACK vars that only occurred in the raw goal. We make an exception for the case where the raw goal comes from the last step in the derivation and the body has exactly one atom; in that case, we do not trim the old θs, since we can prove that no STACK vars can occur in the atom (due to constraint stack_in_TL_body that we added in arg_deriv). This exception simplifies a later stage of the proof.

In stage arg_deriv, the expression that computed the store was the composition of the trimmed θs. In this stage, we replace that expression by the fixed-point of the union of the trimmed θs, where we trim them using the new technique. When we map a fix_deriv derivation to an arg_deriv derivation, the arg_deriv store is the composition of the trimmed fixed-points of the θs. The main proof in this refinement is that the composition of the fixed-points gives the same result when applied to the goal as the fixed-point of the union. This is made very difficult by the fact that the unifiers are trimmed, that is, that some STACK var bindings are removed from the domain of the unifiers.

CHAPTER 22

temp_deriv: use any TEMP you want

One of the original constraints on derivations was that each new clause contain fresh variables; that is what we call the disjointness condition. In the previous refinement we used different ways to enforce this condition for different classes of variables; HEAP and STACK vars were constrained by TOPHEAP, TOPPHI-HEAP and TOPSTACK, ARG vars did not have to be constrained at all (since we were able to prove that they are always fresh), but for TEMP vars we still used the original formulation. In this stage we show that we can simply drop the disjointness condition for TEMP vars as a consequence of the constraints we already have.

We initially did not do this, but rather carried the disjointness condition for TEMP vars along into the following two or three stages of the proof. Eventually it because apparent that it was becoming very cumbersome, so we inserted this stage in between two previously completed ones to eliminate it. This demonstrates one way to correct design errors in the proof; if the error is minor, it can be corrected by changing previous stages, otherwise a new stage can be inserted into the chain.

Note that this disjointness condition for variables is not vacuous. Although our constraints imply that TEMP vars never end up in the goal to be used for resolution, they do not (and should not) prevent TEMP vars from appearing in the stack; the whole point is that they represent variables for which no heap cell has yet been allocated. This means that TEMP vars may end up in the goal list, so on the surface it seems that the disjointness condition would need to be enforced. What we will show is that given a derivation that does not enforce it, we can produce an equivalent one that does.

The idea is as follows. We have already seen that TEMP variables never occur in unifiers, so the only time that TEMP vars are used is when a ϕ is applied to a raw goal. A given ϕ is only applied to an atom in a particular derivation step; therefore, the TEMP vars in all other steps may be changed arbitrarily and it will not affect the result. Therefore any TEMP vars may be chosen for any step.

We now show how to construct a derivation that satisfies the disjointness condition for TEMP vars from a derivation that does not. For this purpose we define a one-to-one mapping from pairs of numbers to numbers called *numsq*; we do not actually give an explicit definition of this function in the proof, but rather only show that such functions exist and use an arbitrary one.[1] For each clause at step m in the derivation, we rename each TEMP var in the clause (with address n) to $numsq(SUC\ m)n$. Since *numsq* is a one-to-one mapping, not only is there is one-to-one correspondence between TEMP vars in the new and old clauses, but the TEMP vars in different clauses must be mapped to distinct vars (since different values of $SUC\ m$ are passed), so the new clauses must satisfy the disjointness condition. It only remains to create corresponding ϕs; the new ϕ runs the mapping backward and applies the original ϕ to the resulting TEMP var, if it is in its domain.

The proofs in this stage are relatively uninteresting. Many properties are trivially preserved because they do not involve either ϕs or the clauses.

[1] Just proving this took us almost a day.

CHAPTER 23

store_deriv: introducing the unification routine

So far our constraints just check that the clause variants, unifiers and ϕs fields have some general properties; they do not say how to create these fields. A WAM actually creates things that correspond to these data structures. This stage introduces two functions that represent in a very abstract way the way the WAM does this; it also introduces different representations for the store and the goal.

23.1 The get and put functions

Many WAM instructions start with the prefixes `get` and `put`. The instructions that a WAM compiler generates in order to match an input goal with a clause head are mostly of the `get` variety, while those it generates to process a body atom are mostly of the `put` variety. We model most of the effect of these two kinds of instruction sequences using two parameters, **get** and **put**. Until now the predicates characterizing derivations have been of the form `XXX_deriv p g deriv`, where p is the program and g is the root goal; from this point on, derivations have the form `XXX_deriv get put p g deriv`.

The whole point of the two functions is to produce some of the fields of the derivation. The **get** function returns four values: a new topheap, a clause, an args substitution, and a unifier. We call **get** once for each derivation step, and use values returned by it as the TOPHEAP, CLAUSE, ARGS and THETA fields. In previous stages we imposed many constraints on these fields; in this step, rather than require only that the particular values used in this derivation satisfy the constraints, we require that (given certain preconditions) *all* values returned by the **get** function satisfy these constraints. In effect, we transfer the constraints from the fields to the **get** function. Similarly, the **put** function returns three values which we use for the fields TOPPHIHEAP, ϕ and the goal; most constraints on those fields are transferred to the **put** function.

As far as the proof goes, there is no particular need for these functions and no particular reason for computing these fields using them; we introduce them here because something like them must eventually be introduced anyway, and the earlier it is done the easier the proof is. Of the fields we have at this stage, it is obvious that TOPHEAP, ARGS and THETA should be generated by **get**; WAM get instructions may raise the value of H, and they may bind values to argument registers or stack or heap cells (which we represent using ARGS and THETA). In contrast, it is unlikely that a typical **get** process would actually construct an entire clause variant, but rather just the clause head. We have it return the entire clause to keep things simple, but we must convince ourselves that we will eventually be able to deliver a **get** function that actually does this.

The **get** (and **put**) process will eventually be divided into a compile-time stage (the WAM compiler) and a run-time stage (the effect of the individual get/put instructions). Obviously, the conceptual combination of compiling a clause head and executing the resulting instructions determines which WAM data structures (argument registers, heap or stack cells) are used to represent variables in the clause head; however any variables that do not occur in the head are not dealt with. Thus, we want the **get** process in this stage to generate the entire clause, but the **get** process we will eventually have in some more refined stage will (probably) not do that. We want to anticipate how we will eventually be able to simulate the abstract **get** process using the refined one. We now show that we would be able to generate the clause body using the both parts of the refined **get** process and the compile-time part of the refined **put** routine (but not the run-time part); that is, the extra information we need about the body is available (for the most part) at compile-time.

We now consider each of the four classes of variables that may appear (only) in clause body. We have already added a constraint that prevents argument registers from occurring only in the clause body, so we need not consider them. If a heap variable only occurred in the body, we would simply have to require the refined **get** process to allocate them; this would mean that at some point the compiler would have to emit instructions to allocate heap cells for these variables and save their value in stack cells. In fact, compilers do not do this, so this case is entirely hypothetical. If the variable is a TEMP var, it represents a program clause variable for which the compiler is going to allocate a heap cell in a delayed fashion. The compiler makes this decision only on the basis of the program clause, so all we would have to do is run the compiler (that is, the compile-time part of the **put** process) on the clause body and use the TEMP vars that it produces. The case of STACK vars is only slightly harder; in this case, we may determine at compile time what stack offsets will be used for the variables, so we may produce the STACK vars using this information together with the value of TOPSTACK. Thus, while we would not ordinarily think of the **get** process as producing an entire clause variant, there is no harm in doing so.

The **get** function we use here does not return anything corresponding to the trail, since the trail is used for backtracking, that is, for the case when unification fails, and we have so far only been considering the case where it succeeds. In order to reach an interpreter, we would eventually have to refine the **get** function to return a trail as well, and require that it meet certain requirements, but we never reach that stage.

We use our **get** and **put** functions to represent sequences of WAM get and put instructions. WAM instructions may be thought of as functions from WAM states to WAM states, so we might expect **get** and **put** to be such functions as well. They are not because we do not yet have a machine state to pass in or return; furthermore, even when we do have things that resemble machine states, we will be refining those states, which would require each stage to define it own (trivially different) version of **get** and **put**. What we do instead is have the **get** and **put** functions take certain fields as inputs and return others as outputs; for the remainder of the part of the proof we have completed, these fields are stored in different places and in different ways, but are conceptually the same, so we may use the same functions.

It remains to choose which fields should be used as inputs. Just as the proof itself gives no guidance as to which fields should be determined by the outputs, so it does not indicate which should be used as inputs. We choose which fields to use as inputs based on what we believe the implementations of the **get** and **put** functions will actually need.

It seems reasonable to have our **get** function take five parameters: the store, the goal, the top of heap, the top of stack,[1] and the program clause. The instructions corresponding to **get** will certainly need these parts of the machine, and perhaps no more. Obviously it needs the goal and the program clause to match it against. The store is necessary in order to dereference variables in the goal. The top of stack is needed to position variables in the new stack frame, and the top of heap is needed if new heap cells are allocated.

23.2 Constraints on get and put

Very little is actually changed by introducing the **get** and **put** functions. Before we were given the TOPHEAP, CLAUSE, ARGS and THETA fields and just checked that they had certain properties. Now we create these fields using the **get** function, and require that the values returned by the **get** function (almost always) have these properties. That is, we transform the properties we used to impose directly on the derivation fields into properties that the **get** function must satisfy. For example, we have always required that the clause of a derivation step must be a variant of the associated program clause; we can now just require that the clause returned by **get** must be a variant of its input clause. Similarly, we used to require that all HEAP vars in the clause variant of a derivation step be greater than the top heap value in the previous step; now we can require that any HEAP var in the clause returned by **get** must have an address that is greater than the top-heap value it is given. Nearly all of the constraints on the data structures that **get** returns (the top heap, clause and the ARGS substitution and unifier) are transferred to the **get** function this way.

[1]The current proof does not pass a top-of-stack value to **get** because we decided to stop using stack variables at this point, but we would have to if we wanted to allow the compiler to use stack cells to represent clause variables.

We cannot simply require that **get** satisfy these constraints for all inputs, though. Obviously, we cannot require that the substitution returned by **get** unify the goal and the head if they are in fact not unifiable. Thus, we may only require that **get** satisfy the constraints if certain preconditions hold.

Currently we have only two preconditions for **get**. The first is that the input goal and clause are unifiable; this we can just require, since we are only going to use **get** to produce fields for unifiable clauses. The other precondition concerns the store. Consider how the WAM dereferences heap and stack cells in the course of the get operation; surely if we cannot assume something about the contents of these cells we cannot even prove that the dereferencing operation (and hence the get) will stop. The obvious precondition for this case is "the store is directed". Since we have already proved that this is an invariant in a prior stage, it is not difficult to add this precondition.

It is certain that other preconditions would have to be made for an efficient concrete **get** routine, however we have not attempted to add more. We believe it is better to add preconditions in a demand-driven fashion, that is, to discover what preconditions are actually needed to prove an concrete **get** routine correct and then show that they hold whenever **get** is called. For example, the proof did not require us to add the precondition concerning directness at this stage, we simply anticipated that it would be needed; we will illustrate how we would deal with other preconditions by considering what would have happened had we not added this one. If we had not added it, then when it eventually came time to write a concrete **get** procedure and prove that it satisfies these constraints, we would have found that we could not even prove that dereferencing would always halt. After we diagnosed the problem and determined that it could best be handled by adding a precondition, we would have guessed an appropriate one that would allow the proof to go through (in this case, "the store is directed"); finally, we would have gone back to this stage in the proof and shown that the new precondition always holds when **get** is called.

All of the constraints that are now imposed on the **get** routine are collected into one predicate called `is_awam_get`; these constraints are eventually transferred to the compiler and the get instructions. All the remaining stages in our proof use a **get** function that satisfies this predicate; the only difference is where the parameters come from and where the returned values are stored.

The **put** function is similar to the **get** function, but has more complex constraints. It takes seven parameters: the raw goal to be put (that is, a variant of a program body atom), an args substitution, a store, a cphi, two predicates called in_head and in_body (explained below), the index of the atom being put, and the top heap. It returns three values: a top heap, a ϕ, and an args substitution.

We now justify why these inputs and outputs completely model the action of the **put** routines together with the constraints on the way the WAM compiler uses them. Obviously, the compiler needs to know which body atom is being processed. In this stage we still use atoms in the body of the clause variant, not the program clause itself (and we never get to that in our proof). We believe that this is an acceptable abstraction. The variant contains the particular heap and stack cells and argument registers chosen by the compiler to represent variables in the program clause (as well as TEMP vars that represent variables the compiler will use delayed allocation for); it would be an unnecessary complication to deal with these issues at this point. The args substitution is only used if the atom being processed is the first atom in the body. After the head has been processed by the **get** routine, some values may have been loaded into argument registers; we represent this using the args substitution returned by the **get** function. When processing the first body atom, the **put** routine may use these values to construct the goal; that is why we pass the args substitution to the **put** function. However, it is not needed for any subsequent body atom, so in those cases we pass the empty substitution (which essentially gives no information). The WAM **put** routines may dereference cells, so we must pass the store to the **put** function. Some variables in a clause body may not be allocated heap cells until the first atom in which they occur is processed (we represent these using TEMP vars at this point). If such a variable occurs later in the clause, then the heap cell allocated for it must be saved. Since we have represented this using the cphi mapping, we must pass it to the **put** function. The actual WAM represents this mapping using cells in the stack frame, but it would complicate the proof unnecessarily to do that here. The parameters in_head and in_body are used to deal with body variables. The in_head parameter tells the **put** routine whether or not a particular variable occurs in the clause head; it needs to know this since it only takes special action for variables that do not occur in the head. Similarly, it needs to know if this is the first or last occurrence of the variable in the clause body; the in_body parameter allows it to do so, since it is a list of predicates with exactly as many elements as there are body atoms, each of which says whether or not a variable occurs in the corresponding body atom. Finally, the th parameter is needed in order to allocate fresh heap cells. While these values are all abstract, there is a close correspondence to real WAM data structures, yet they are easier to reason about because of their abstractness.

The **get** and **put** functions make the addition of a new resolution step almost, but not quite, deterministic. Obviously CLAUSENO cannot be determined. What is more interesting is that TOPSTACK cannot, although we have required that **get** determine TOPHEAP and **put** determine TOPPHIHEAP. The difference is that the heap is used for nothing else but representing terms, either the initial goal or terms resulting from the get and put operations. In contrast, the WAM stack is used for several different things, several of which are not yet represented at this stage, so we cannot provide the **get** function with enough information to produce a value that will be acceptable in later stages.

23.3 Representing the goal using GOALARGS

Until now, a raw goal has been explicitly represented by an atom; we then applied a complex substitution (involving the store, ϕ, cphi and args) to that to get the goal. There is no explicit representation of the raw goal in the WAM; instead, the argument registers point to what may be thought of as the terms in the body of the raw goal. We may think of this as a substitution from ARG vars to terms, which we could apply to the appropriate atom to get the raw goal. The atom we apply it to must have the same functor as the goal atom, the same number of body terms, and an ARG var with address i for the ith body term.

There is a complication here that we do not address. The contents of the argument registers cannot be put into direct correspondence with the raw goal, since in some cases a **put** routine may load the dereferenced value of a cell into the argument register, not the cell itself (which would represent a variable in the raw goal). This means that we cannot say that the result of applying this substitution to the template atom gives the raw goal, so we have said instead that the result is the goal itself, that is, it also takes the effect of the store into account. This may or may not turn out to be a problem when one attempts to show that an actual **put** routine satisfies this constraint.

23.4 Representing the store

Until this stage we have called a complex combination of the derivation unifiers "the store". The WAM does not keep the past unifiers separate and combine them for each resolution step, but rather has a (conceptually) single data structure (divided into the heap and the stack) that contains this information. In this stage we show how to express the store using a recursive function of the derivation. In each step, the new store is constructed using the old store and the new unifier; we show that the new method of computing the store gives exactly the same result as the old method.

CHAPTER 24

bodyno_deriv: using an explicit bodyno

Until this point we have used a function to determine which body atoms of each clause in the derivation are left to process. That is, *deriv_bodyno deriv* i tells us how many body atoms from step i have already been processed, and hence also how many are left. In contrast, the call $deriv\_bodyno(FIRSTN\ deriv\ n)i$ tells us how many body atoms from step i had already been processed for the subderivation of length n (where $i < n$). This is how we were able to represent the raw goal lists for all subderivations of the derivation, which made proofs simpler, since we only needed to induct on the derivation.

All we do in this refinement is represent the bodyno explicitly at each resolution step by adding a counter. Now, to determine how many body atoms have already been processed at step i, one examines the counter at step i rather than call *deriv_bodyno deriv* on i. However, since counters are changing, we can no longer just add resolution steps to the derivation; instead of just a list of resolution steps, we must now use a list of lists of resolutions steps. To add a new step, we must now copy the list from the previous step, increment the appropriate bodyno counter, append a new resolution step to the list, and append this new list to our derivation. The proofs in this stage now run over two indices, one for the list of derivations, the other for the steps within a derivation; this makes the proofs large and tedious.

This may be made clearer by re-examining figure 15.2 in chapter 15. In the old style, there was only one data structure, but by applying the function to different prefixes we could get the bodyno positions for any previous stage in the derivation. Now these bodyno positions are stored explicitly; the first resolution step contains the first step of upper left diagram, the second step of the new derivation is the first two steps of the next diagram, the third is the first three steps of the third diagram, and so forth.

The derivation constraints must be changed to reflect the new style, but mostly in minor ways. Despite the fact that this is a conceptually simple change, this is one of the larger stages of the proof.

CHAPTER 25

ds_deriv: collapsing the stack

It should be obvious that many parts of a derivation are only needed for a short time; for example, the ARGS substitution of a given resolution step is only used for the immediately following step, if at all. One only needs a part of the entire derivation to "continue" it, and indeed WAM interpreters discard much of what is contained in a derivation. This refinement discards most of this useless information.

A derivation in the previous stage was a list of lists of resolution steps; each of these lists is (roughly speaking) a derivation, except that it explicitly encoded which body atoms had been processed. In this stage we discard much of the information in each of these lists and put the rest in a new data structure called a DERIV_STEP, so our new derivation is a list of these new data structures. Roughly speaking, a DERIV_STEP will contain some fields from the last step of the list, together with some of the fields of some of the prior steps.[1] The fields that we take from the last step are: the topheap, topphiheap, topstack, unifier, args, goalargs, phi, store, clause and clauseno. Only a few fields are needed from prior steps; they are gathered together in a new data structure called DERIV_STACK_FRAME, which has the following fields: clauseno, body, bodyno,

[1] Keep in mind that these are steps in the list contained in the last resolution step of the derivation.

phi, topstack, in_head, in_body. The phi field in the stack frame corresponds to the cphi field in the previous stage; in_head and in_body were computed in the last stage, but are now explicitly stored; they eventually become symbol table attributes.

Not only do we only use some of the fields of prior resolution steps, but we discard all prior steps that have no more body atoms left to process; that is, we only keep those steps that were part of what we called the "stack". The corresponding list of DERIV_STACK_FRAME elements begins to resemble the WAM stack.

In previous stages we determined where the next goal was by searching down the derivation looking for the first resolution step with body atoms left to process. In this refinement we are more efficient, in that we don't have to search for the next goal, but the price is that we must deal with specific cases. There are four possibilities, depending on how many atoms are left to process in the body of the current clause and the body in the top stack frame (the current clause is the clause just matched by a **get**operation). We call a resolution step a

- **call_goal** if the body in the current clause has more than one atom
- **chain_goal** if the body in the current clause has exactly one atom
- **leaf_call_goal** if the body in the current clause is empty and the body in the top stack frame has more than one body atom left
- **leaf_chain_goal** if the body in the current clause is empty and the body in the top stack frame has exactly one body atom left

We prove that all the body in all stack elements must have at least one atom in it, so these four cases are exhaustive. We will now discuss why these cases are sensible, and indeed necessary.

If the body of the current clause has more than atom we will not be able to finish it in just one resolution step; we must save the second and subsequent atoms in the body so that we can process them later (the first is used as the goal in the next resolution step). This requires a new stack frame. Since this is the only way a frames is ever added to the stack, we can easily prove that all stack frames have a non-nil body. If the body of the current clause has exactly one atom, no atoms need to be saved, so the stack need not change at all. If the body of the current clause is empty, then we ask the same questions of the body on top of the stack; if it has more than one atom left, then we change the top frame by popping the first atom off and continue, otherwise it must have exactly one atom, so we pop the frame off and use that atom as a goal.

In all four cases the current unifier is unioned into the current store to create the new store, but there is a minor difference between the chain_goal and the other cases. In previous stages we have already required that if a clause contains STACK vars, then they must appear in the second or subsequent body atoms. The clause of a chain goal cannot satisfy this constraint, because its body must contain exactly one atom; therefore, chain goal clauses cannot contain STACK vars. This means that there is no need to trim STACK vars from the store. We will see that this is important in the next stage. In the case of leaf goals, we pop a parent body atom, which may contain the last occurrence in that body of a given STACK var, so any bindings for that STACK var in the store must be removed. In the case of call goals we in fact never need to trim STACK vars, since the first atom of the current clause is used as a goal, and it cannot contain

	Old stack	New stack
Call goal		ngs
	(cg@cgs)	cgs
Chain goal	[cg]	ngs
Leaf call goal	[]	ngs
	(pg@pgs)	pgs
Leaf chain goal	[]	
	[pg]	ngs

Figure 25.1: The four possible ways in which the stack may change from one resolution step to the next. In the first case, the current goal has more than one atom (cg@cgs) so the first one is popped and resolved with a clause, giving a new list of atoms (ngs). In the second case, there is only one atom in the body of the current goal, so it is resolved giving a new list. In the remaining cases the current goal has no body; if the parent has more than atom (pg@pgs), then the first one is resolved and the rest remain, otherwise the frame with the one remaining parent atom is replaced by a frame containing the new atoms.

the last occurrence of any STACK var, but we will see that this is not important for subsequent stages.

CHAPTER 26

op_deriv: introducing transitions

The last refinement introduced a data structure that is recognizably WAM-like, however, the difference between the state in one step and its successor is much greater than the difference created by a WAM instruction. In this refinement, we begin representing a resolution step with a sequence of states rather than just a single one. Each state in the sequence is (roughly speaking) the result of applying an instruction to the previous state in the sequence; the first state is the result of applying an instruction to the last state in the sequence of the previous step (if there is one). The difference between one state and the next is now roughly similar to that caused by certain WAM instructions.

The instructions in this refinement are called: ds_allocate, ds_deallocate, ds_proceed, ds_execute, ds_call, ds_get, and ds_put. The first five correspond to the WAM instructions of the same name; the last two are an abstraction of the effect of execution all the WAM instructions needed to unify the head with the goal and to put a goal atom, respectively. We continue to use the **get** and **put** function parameters introduced in an earlier refinement. Therefore, we do not yet handle the details of unification, nor do we address the issue of choicepoints.

In the previous refinement we showed that there were four ways in which one state may differ from the next one; we called these four possibilities call, chain, leaf_call and leaf_chain. The differences stemmed from how many body atoms the current and parent clauses still had left to process. In this refinement there are still four different alternatives, but they are expressed as four different kinds of state sequences. The four different execution sequences will have the same names, but begin with the prefix "ds_".

Each of the four sequences begins with a ds_put instruction and ends with a ds_get instruction. This may seem odd for those experienced with the WAM, because the code that a WAM compiler produces first deals with the clause head (corresponding to a **get**) and then processes the body atoms (corresponding to a sequence of **put** and call instructions). However, when seen from the point of view of a resolution step, our sequence makes more sense. A new resolution step first chooses a new goal to resolve against and then attempts resolution.

The sequences differ in how they treat the stack, that is, whether they push or pop frames from it, and whether or not they save information in the topmost frame (for a call). For simplicity, we always push a new stack frame on top of the stack for every new goal, whether or not it really needs it.

A ds_call_goal has four states that correspond to the following sequence of instructions: ds_put, ds_call, ds_allocate and ds_get. Recall that this case is used when the current clause has more than one body atom; therefore the call instruction does not pop the top stack frame, which corresponds to the current clause. The allocate instruction is called simply because we always allocate a new stack frame in this refinement.

A ds_chain_goal differs only in that it calls ds_execute rather than ds_call. The execute instruction must pop the top stack frame, because it corresponds to the current clause and it is no longer needed.

A ds_leaf_call_goal corresponds to this sequence of instructions: ds_proceed, ds_put, ds_call, ds_allocate and ds_get. Recall that in the last two of the four cases the current clause has no body; the ds_proceed instruction reflects this by popping the stack. In the WAM, of course, the proceed instruction simply transfers the contents of the cp register to the p register, it does not affect the stack. At this point, however, we do not yet have program counters, and a new top element is always pushed on the stack even if it isn't needed, so it must be discarded at this point.

Finally, a ds_leaf_chain_goal uses this sequence of instructions: ds_proceed, ds_put, ds_deallocate, ds_execute, ds_allocate and ds_get. In this case the parent has only one body atom left to process, so we pop that stack frame immediately after processing that atom (just as the WAM does). The execute instruction for chain_goal pops the stack, but this execute instruction does not; the stack is popped by the call to ds_deallocate.

The design of the instructions is intended to resemble WAM instructions, but it also is intended to simplify the proof of equivalence. Some features of the WAM are delayed until subsequent refinements.

For simplicity, we always push a stack frame. The topmost stack frame contains (among other things) the body of the current clause. The put instruction operates on the topmost stack frame; therefore, if the topmost stack frame has an empty body (which is the case in the last two paths) it must be popped before ds_put is called.

In previous stages there were two substitutions that represented the argument register; GOALARGS represented the contents of the argument registers just prior to unification, while REGS represented their contents afterward. That was necessary because we had to represent both values in one state. Now we have a sequence of states, so we can use one substitution to represent both; we use the contents of DS_REGS just after the put instruction as GOALARGS and the contents of DS_REGS just after the get instruction as REGS.

CHAPTER 27

pc_deriv: adding program counters

In this chapter, we add program counters and store them on the stack the way the WAM does.

In the previous refinement, control flow was determined by examining the current clause and possibly also the topmost clause body on the stack. In a real machine this would correspond to examining the data structure that represents the clause and the body; this is not what the WAM does, of course. We introduce the concept of a program counter and use two of them to represent the current clause and the topmost clause body. We use a convenient abstraction for program counters; it is a tuple with the following fields: clauseno, clause, bodyno, and modifier. The clause is the actual program clause being used; it is convenient to put that into the counter at this point. The clauseno is just the index of this clause into the program, to uniquely identify it. The bodyno is a natural number indicating how many of the body atoms have already been processed. Finally, the modifier indicates which kind of instruction should be executed next.

There are two difficulties in relating this refinement to the previous one. When a call is made, the current cp is stored in the new frame, just as in the WAM; when that frame is deallocated, the cp is restored from that saved value. This means that a program counter is stored in each stack frame, but this program counter is *not* associated with the body in that frame, but rather with the body in the frame below it. The cp is associated with the body on the top of the stack; the program counter in the top frame is associated with the body in the frame below that, and so forth. Therefore, to create an old-style stack from a new-style one, we must shift these program counters down by one element.

The second difficulty is that we no longer push a stack frame for leaf procedures and chain rules. In the previous refinement, to keep things simple we always allocated a stack frame; now we don't, so to create an old-style stack from a new-style one we must sometimes add another frame on top.

CHAPTER 28

cp_deriv: adding choicepoints

A derivation is a list of resolution steps; each step is by definition a successful unification. A real machine will not always be successful when attempting a unification; it maintains extra state in order to be able to try other alternatives in case of failure. In this (final) refinement we allow the derivation to add this extra state to support backtracking, but we do not actually define a backtrack operation and hence never reach an actual working machine.

Roughly speaking, when a procedure with more than one clause is called, the WAM pushes a data structure called a choice point onto the stack before it tries the first clause. It stores enough information in the choicepoint so that in the event that the attempt to use the first clause fails, the effects of the attempt can be undone and the second clause can be tried (and after that the third, and so forth). In this refinement, we create a new data structure that is an abstraction of a choice point and modify the four possible resolution paths so that a choice point may be added. We never need to require anything in particular about whether or why choicepoints are allocated; we simply show that the new style of derivation corresponds to the old one whether or not they are allocated.

The main difference resulting from the addition of choicepoints is how the stack is viewed. In the previous refinement, if the current clause has no body, then the topmost stack element contains the next atom to resolve against. With the addition of choicepoints, this is no longer the case; instead, we use a new register (E) to point to the frame in the stack where the next atom is, and this frame will have an E field that will point to the next element down, and so forth. The proof of equivalence largely concerns showing that one can recover an old-style stack by picking out the appropriate elements of the new one. Choicepoints themselves are never actually used, it is just shown that they do not interfere with the existing mechanism.

CHAPTER 29

The result of the proof

This chapter sketches the AWAM data structure and how it is used in our last refinement stage to give the reader a concrete idea of what was accomplished. The result is far too complex to describe in detail to those who are not already familiar with the WAM; the intent is to describe enough so that those who are familiar with it can recognize its features. We describe three data structures: the AWAM and the abstract representation of environments and choicepoints. This is followed by a description of the cp_deriv predicate, and a description of two instructions modeled after the WAM call and get instructions.

Table 29.1 contains the fields of our abstract environment. Fields ENV_CP and ENV_E correspond directly to fields in a WAM environment: ENV_CP is the saved value of the CP register, and ENV_E is the saved E register.[1] The remaining fields will seem strange. ENV_PHI is the mapping from TEMP variables to HEAP variables corresponding to the collected ϕ mappings in previous stages; it records which heap cells have been allocated for on-the-fly initialization of body variables that were never represented using stack cells. Eventually, this mapping would be implemented using stack cells; the compiler would assign every such body

[1] In our abstract machine the E register is the index where the current stack frame occurs in the stack. This is a little different from the WAM E register, which both points at the current stack frame and helps determine where new stack cells are allocated (along with the B register). At this level of abstraction we still do not represent the stack using stack cells, so a top-of-stack value cannot be used to locate the top stack frame; therefore we use two values.

variable a stack cell, and the first time the variable is encountered a heap cell would be allocated for it and saved in the stack cell so that it could be used when the variable appears again. ENV_PHI is trimmed when the call instruction is executed, which models environment trimming of these stack cells. ENV_BODY contains the body atoms that still have not been processed in this frame; although it isn't usually thought of this way, most of this information can be recovered from a WAM stack frame. ENV_IN_HEAD is a function that returns true iff the variable it is passed occurred in the head of the clause variant that was created when this stack frame was allocated; ENV_IN_BODY is similar, but it is a list of functions, each of which determines if a variable occurred in a particular atom of the body. When compiling a clause, a WAM compiler treats a variable specially when it encounters the first or last occurrence of a variable that only occurs in the clause body. In order to do so, it must first scan the entire clause before generating code; the ENV_IN_HEAD and ENV_IN_BODY functions represent this action of the compiler.

The fields of our abstract choicepoint are given in table 29.2. These fields are less abstract than those of ENVIRONMENT. When a choicepoint is created, various parts of the abstract machine are saved in these fields, just as the corresponding fields are in the WAM.

The fields of the AWAM data structure are given in table 29.3. Most of the fields directly correspond to the WAM, with the exception of the clause, store, stack and trail fields. The AWAM_CLAUSE holds the clause variant created by the call to the abstract **get** function; in fact, an actual WAM only creates a variant for the clause head at the corresponding point in execution, but as we explain in chapter 23, we believe that this is a reasonable abstraction. The AWAM_STACK

ENV_CP	the saved continuation pointer
ENV_PHI	the mapping from TEMP variables to HEAP variables
ENV_BODY	the remaining atoms left to process
ENV_IN_HEAD	which variables were in the head
ENV_IN_BODY	which variables were in each clause body atom
ENV_E	an index into the stack pointing to the parent environment

Table 29.1: Fields of the ENVIRONMENT data structure.

CP_TR	pointer to the top of trail
CP_ARGS	saved argument registers
CP_B	saved B register
CP_E	saved E register
CP_TOPHEAP	saved TH register
CP_CP	saved continuation pointer

Table 29.2: Fields of the CHOICE_POINT data structure.

is a stack of ENVIRONMENT and CHOICE_POINT data structures, so it is not just a linear array of stack cells as in the WAM; that would be implemented in a later refinement. The AWAM_STORE holds a substitution that represents the contents of the WAM heap and some of the WAM stack cells; these stack cells have nothing to do with the AWAM_STACK at this point. The fact that stack cells are eventually used both to implement stack frames and to represent variables does not matter here. Finally, the AWAM_TRAIL is an abstract representation of the trail, but it was still not developed much at this point.

The last stage defines the cp_deriv derivation type; the corresponding predicate is given in figure 29.1. A derivation is a list of resolution steps; in this case, we name the derivation "awams_list", each element of which is a list of AWAM data structures. The predicate cp_deriv determines if awams_list is a valid derivation from program p and initial goal g; it uses the given abstract functions get and put to test this. The first two conjuncts ensure that get and put satisfy the constraints that we developed for data representation; these are described in chapter 23. The fourth conjunct ensures that all variables in the initial goal are heap variables; the third ensures that all of these variables have addresses less than the initial top-of-heap th. The sixth conjunct ensures that the AWAM states are execution sequences. The SNOC_PROP6 function applies the closure `cp_bodyno_ok get put needs_cp` to every adjacent pair of resolution steps; the first argument passed to the function is the last AWAM in the one resolution step, and the second argument is the list of AWAMs in the next resolution step. Thus, in the definition of cp_bodyno_ok, awam0 represents the last AWAM in one resolution step and awams represents the list of AWAM states in the following resolution step. The list is an acceptable resolution step if it is one of four kinds

AWAM_P	program counter
AWAM_CP	continuation pointer
AWAM_TOPHEAP	H register
AWAM_ARGS	argument registers
AWAM_CLAUSE	clause variant created to match the goal
AWAM_STORE	heap and stack cells used for the store
AWAM_STACK	stack frames and choicepoints
AWAM_B	pointer to top choicepoint
AWAM_E	pointer to top stack frame
AWAM_TRAIL	trail

Table 29.3: Fields of the AWAM data structure.

of execution sequences (these four already appear in stage ds_deriv, as described in chapter 25). A sequence of AWAM states is a valid cp_call_goal sequence if it is the result of executing the following instructions: put, call, try, maybe_cp_allocate and get. The pattern of put/call/get is common to all the sequence types, with small variations. The first conjunct of cp_call_goal ensures that this sequence may only be used if the program counter contains a clause that has more than one body atom, otherwise one would have to use one of the other sequences.[2] The symbol "?" represents existential quantification in HOL; cp_call_goal does not completely constrain the program counter used for the start of this sequence, to allow the resolution step to non-deterministically choose which clause to use for resolution. Finally, several of the instructions take abstract functions, so for example the function that implements the get instruction calls the abstract **get** function (that is passed as an argument to cp_deriv) to create some of the fields of the new AWAM.

We now describe two instructions, one simple and one complex. The simple one is cp_call, shown in figure 29.2. All instructions take an AWAM state as input and return an AWAM state as output, just as WAM instructions in effect do by modifying the WAM state. We use a pattern-matching style in our definitions, so for example the program counter of the AWAM state passed to cp_call is p; the order of the AWAM fields is the same as given in figure 29.3. The cp_call instruction takes a program counter argument p', which becomes the returned AWAMs program counter.[3] As in the WAM, the current program counter is saved in the

[2]For convenience, our abstract program counters actually contain the program clause rather than just being an index into some kind of code; later on this would be eliminated using partial evaluation.

[3]PC_SET_MODIFIER sets a field in the program counter; in this case, it indicates that the next instruction will be a get.

```
cp_deriv get put needs_cp th p g awams_list =
is_awam_get get /\ is_awam_put put /\
(!v. atom_subvar v g ==> var_address v < th) /\
(!v. atom_subvar v g ==> var_is_heap v) /\
SNOC_PROP7 (cp_calls_ok get put needs_cp p) awams_list /\
SNOC_PROP6 (cp_bodyno_ok get put needs_cp)
              (cp_root_ok get th p g) awams_list


cp_bodyno_ok get put needs_cp awam0 awams =
cp_call_goal get put needs_cp awam0 awams \/
cp_chain_goal get put needs_cp awam0 awams \/
cp_leaf_call_goal get put needs_cp awam0 awams \/
cp_leaf_chain_goal get put needs_cp awam0 awams


cp_call_goal get put needs_cp awam0 awams =
?pc. SUC 0 < LENGTH (clause_body (PC_CLAUSE(AWAM_P awam0))) /\
     valid_next_pc (AWAM_P (HD awams)) pc /\
     (do_steps awam0 [cp_put put; cp_call pc;
            cp_try needs_cp;
            maybe_cp_allocate; cp_get get] = awams)
```

Figure 29.1: The predicate defining cp_deriv and some supporting routines.

```
cp_call p' (ABSWAM p cp th args clause store stack b e trail) =
let body = ENV_BODY (EL (lower e) stack)
and top = EL (lower e) stack
in
ABSWAM (PC_SET_MODIFIER PC_GET p')
        (lift (INC_BODYNO p))
                th args clause
        (restrict_P (not_stack_in_hd1 body) store)
        (REPLACE_NTH (lower e) cp_call_env stack)
        b e trail
```

Figure 29.2: The definition of the call function.

returned AWAM's CP register.[4] Most of the rest of the state remains unchanged, with the exception of the store and the stack. In a WAM that implements environment trimming, the stack pointer may be reduced when a call is made; this is why the store is trimmed (by the call to restrict_P). The current stack frame is pointed at by the E register; this frame is modified slightly (by the call to REPLACE_NTH), also to reflect environment trimming.

We now describe the get instruction, shown in figure 29.3, as an example of a complex instruction. This takes the abstract get function as an argument and calls it (via the function call_get, shown in figure 29.4) to create new fields (which are stored in the structure ginfo). The local variable "clause" is the variant created by get. The current pc is incremented; depending on whether

[4] We use a lifted type for the cp field because sometimes the continuation pointer is undefined. INC_BODYNO sets a field in the program counter.

```
cp_get get (ABSWAM p cp th args clause store stack b e trail) =
let ginfo = call_get get
                (ABSWAM p cp th args clause store stack b e trail)
in
let clause = AWAM_GET_CLAUSE ginfo
in
ABSWAM (PC_SET_MODIFIER ((clause_body (PC_CLAUSE p) = []) =>
            PC_PROCEED | PC_PUT) p) cp (AWAM_GET_TOPHEAP ginfo)
        (AWAM_GET_ARGS ginfo)
        clause
        (union store (AWAM_GET_THETA ginfo))
  ((stack = []) => [] |
   pc_needs_env p =>
        let top = LAST stack
        in
        SNOC (ENVIRONMENT (ENV_CP top) (ENV_PHI top)
                (clause_body clause)
                (check_atom_subvar (clause_head clause))
                (MAP check_atom_subvar (clause_body clause))
                (ENV_E top))
                (BUTLAST stack) |
   stack)
        b e (APPEND trail (AWAM_GET_TRAIL ginfo))
```

Figure 29.3: The definition of the get instruction.

```
call_get get awam =
   (get (AWAM_STORE awam) (AWAM_ARGS awam)
        (AWAM_TOPHEAP awam)
        (PC_CLAUSE (AWAM_P awam)):awam_get_info)
```

Figure 29.4: The definition of the call_get function.

the current clause has body atoms, the next instruction may be put or proceed. The new args and clause fields are created by get, and the part of the unifier represented by stack and heap cells is added to the store. The computation of the new stack is complex. The borderline case where the stack is empty requires special treatment for reasons we shall not describe. Otherwise, if the clause being processed needs an environment, that is, if it has more than one body atom, then a new stack frame would have been allocated before get was called (by the maybe_cp_allocate instruction) and we now need to modify it; we load the body of the new clause and the functions that record which variables were in which parts of the clause. The loading of the clause into the stack frame models the changes made to stack cells during the get process; as described above, the ENV_IN_HEAD and ENV_IN_BODY functions eventually become part of the compiler. Finally, we append the trail component that get created on the AWAM trail.

CHAPTER 30

What remains to do

A large amount of interesting work is still needed to continue this proof all the way to a low-level machine like the WAM, however, we believe that no insurmountably complex problem would arise to prevent its completion. This chapter gives a brief description of what still needs to be done.

Stage cp_deriv proved that choicepoints do not interfere with the resolution mechanism; the next stage would correspond to the third of the three parts of chapter 2. The target machine's state would be an AWAM data structure; the meta-machine's state would would be a cp_deriv derivation. The proof would be similar to the one sketched for TrivInt at the end of chapter 2. This target machine would still use abstract **get** and **put** functions. The purpose of doing the refinement would not be an end in itself, but rather to understand how the proof would go and see if there are any hidden difficulties. It may be prove to be better to refine representation details before this final stage, in which case one could adapt this proof to use the refinements. It was always difficult to predict what future stages would look like, and stages generally changed quite a bit as the proof proceeded; one must simply forge ahead and adapt to conditions as they arise.

A large amount of low-level detail remains to be implemented. Although constraints on variables have been worked out and can be simply translated into constraints on cells, we have not represented structure functors and constants. It isn't clear at what point these should be modeled. One might do it after control has been refined, or one might refine terms and substitutions so that they use something like cells before control is refined. Stack frames and choicepoints need to be represented using stack cells; furthermore, the machine often keeps track of heap cells by having stack cells to point at them, and these stack cells have not yet been represented.

When these new uses of stack cells are introduced, one will have to use the theorem (first proved in stage trim_deriv) that the store is closed, that is, that all variables in the store point to variables that are also in the store; this will be needed to prove that the machine does not corrupt stack frames and choicepoints as it executes. We suspect that this may unfortunately not be enough; at some point the refined machine will have more steps (to represent individual get instructions, for example), and the inherited theorem concerning the closure of the stack will not apply to these new states, so one would in effect have to prove it again. It may not be worth the effort of propagating this particular property. On the other hand, one may be able to start representing the smaller steps earlier on; perhaps the abstract **get** function could return a sequence of state updates rather than just one. It remains to be seen what approach works best.

For convenience we have assumed that substitutions are infinite, so that we could use the builtin HOL function notation to represent them. At some point one would have to prove that the particular substitutions we use are in fact all finite, so that they can be represented using a finite store.

We have proved the correctness of a unification algorithm modeled after Russinoff's, but we never reached the point where we could add it to our machine. It is unclear whether it would be worthwhile to first add an abstract version (that uses terms and substitutions) and then refine it into a low-level one (that uses cells), or if one should just add a low-level version. This is an issue of practicality that one would simply have to experiment with.

In the language of chapters 11 and 12, we spent most of our time proving the Y_2_X theorems, that is, we proved that refined derivations may be mapped into more abstract ones; we delayed proving the converse (that for every abstract derivation there is a refined one that maps onto it) in the belief that those proofs would be easier. This was true for the most part, with one exception. It turned out in stage store_deriv that it was difficult to transfer the constraints on stack variables to the **get** and **put** functions. We believe that the general framework for proving the equivalence of stages may have to be made more complex to deal with this case, and we simply did not have the time to do it, so we stopped using stack variables at that point. This would have to be fixed in order to model an efficient LP interpreter.

In our model, if a body variable is represented by a stack variable, it may become unsafe because of environment trimming. Alternatively, one may use stack variables and not trim them until after the last atom in which they occur has been processed; in this case they can never become unsafe, although if the variable occurs in the last stack frame it will prevent the frame from being released early. This would be a challenging feature to add.

Finally, at some point one would use partial evaluation to separate the interpreter into compile-time and run-time components, that is, the compiler and instruction set. The design of the stages leading up to this is intended to make that process simple, but it would probably still require some readjustment.

CHAPTER 31

Retrospective

The original problem the author intended to solve three years ago was to mechanize the paper proof published by Russinoff. After working for a year on Russinoff's proof, we had only reached section 4.5 and had not even proved everything prior to that. Lemma 4.5 (which is in an earlier section) had initially been incomprehensible, but as we proved the theorems leading up to it we learned enough to start it (we never fully completed that proof or the others later in that section). After working on that proof for several months, we expected to learn enough to start the ones in section 4.5; that never happened. We decided that we would not be able to finish the proof in a reasonable amount of time, if at all, so we gave up that year's work and began again from scratch.

The fact that we failed to understand Russinoff's proof certainly does not mean that it is bad or cannot be mechanized. It is very likely that someone with outstanding ability (such as Russinoff himself) could do so. However, we believe that our experience indicates that an average WAM implementor (who may not be experienced in theorem proving) will also not be able to understand it, and it seems desirable to find a proof that an implementor can understand.

We did not initially intend to rely heavily on derivations; indeed, we use them only because Russinoff uses pure Prolog as his source language, and hence uses derivations to define its semantics. We assumed that we would quickly arrive at a machine-like object, albeit one much more abstract than the WAM. We were surprised to discover that it would take many more stages than we originally anticipated to reach our goal. After proving the equivalence of the first few intermediate machines, we realized that reasoning about backtracking was annoying and always looked the same. It took some time to realize that most of what we were doing was just mapping one style of derivation to another.

We initially thought that we would be able to start with a general *unify* operation and be able to split it into **get** and **put** components in a later stage. It took a fair amount of time to realize that on-the-fly allocation of body variables makes this difficult. We introduced the ϕ mapping as a temporary measure to help introduce **put**. During this time we realized that things would be easier if we concentrated on derivations rather than machine execution; this meant it was less important to use functions to generate fields of the derivation, so we no longer felt the need to introduce **put** early on. We originally did not introduce **get** and **put** until we first did stage pc_deriv, but at that point it became clear that it would have been easier to introduce them earlier, so we pushed their introduction back into stage store_deriv. This pushing back-and-forth of concepts and constraints between stages of the proof was not unusual.

Inventing and proving the necessary constraints for unsafe stack variables was a nightmare.

It had always been a concern that our proof concerned pure Prolog, and thus that it may not be considered relevant by the LP community in general, since pure Prolog is not considered practical. It took some time to see that the concept of derivation is not that closely tied to pure Prolog, and that if data representation really is as difficult as we believe it is, then there is no reason derivations could not be used to solve this problem in other LP languages as well.

Although we estimate that it would take a year to complete this proof, it may in fact take much longer. Every aspect seemed to take more work than anticipated; just proving the necessary theorems about the abstract tree traversal mechanism took about three weeks, at least another two or three weeks were spent on theorems involving set theory, a day was spent proving that an injection from pairs of natural numbers to natural numbers exists, etc. We recommend that future researchers restrict their first attempts to much simpler machines than the WAM in order to be able to produce a complete proof. The following implementation techniques should be added in this order only after a proof for a machine without these features has already been completed.

- representing variables in the clause head with argument registers
- representing variables that occur in the clause head with stack cells
- on-the-fly initialization of body variables (but not of unsafe stack cells)
- unsafe stack cells

It may seem extreme to recommend that all variables be represented by heap cells in the initial machine, but we believe that the complete task is difficult enough to justify this simplification.

The task of proving the correctness of "the WAM" is open ended. There are many variations of the basic idea, and many proposals for alternative abstract machines[17]. It is likely that most if not all of these machines will share many common features. After a correctness proof for one of the simpler variations has been implemented, the next challenge will be to create refinement proofs of several machines that share as many stages as possible, highlighting the similarities and differences. In a sense, the proof will define a taxonomy of LP interpreters. Much exciting work remains to be done.

CHAPTER 32

Appendix

This appendix gives a detailed explanation of the problem we see in Börger and Rosenzweig's proof of the Pure Prolog Theorem, which was mentioned in chapter 8.1.

The *rename* function is defined on page 35 of [5]; it takes a term and an integer (the variable-renaming index) as arguments. When this function is called by their rules, it is passed the index *vi(currstate)*, where *currstate* is the current state of the interpreter, and the function *vi* is a function that may be thought of as extracting a component of the state (see the definition of *r_code* on p. 41). The global variable *vindex* is distinct but related. Unlike *vi(currstate)*, it is global (p. 40); it is initialized to one and is incremented each time a new goal is processed.[1] In preparation for processing a new goal, one backtrack node is created for every candidate clause (the **selection** rule on page 43); a node is just a machine state, so one possible machine state is saved for each alternative.[2] The *vi* component of each of these nodes is initialized with the current value of *vindex*, hence they all have the same value. The clause used for unification is determined by the equation *r_code* $= Unify(G)$ in the **Unify** rule (p. 43).[3] The symbol *r_code* is

[1] At this stage, backtracking is implemented by simply setting the current state to a previous one. Thus, *vi(currstate)* is affected by backtracking, but *vindex* is not.

[2] The machine state itself is a tree.

[3] Here, *r_code* provides a value and *Unify(G)* is just a pattern-matching expression that allows G to be used elsewhere in the rule.

defined in terms of the function *rename* and the value of *vi(currstate)* (p. 41).

Although is isn't explicitly stated, in order for this proof to work, *rename* must produce terms with distinct variables when given distinct arguments:

$$\forall tuij. i \neq j \vee t \neq u \rightarrow vars(rename(t,i)) \cap vars(rename(u,j)) = \emptyset$$

It must do this, because the clause variant used for resolution may not share any variables in common with the current goal list. Since no information specifically concerning the goal list is passed to it, the renaming function must satisfy this stronger condition. This is perfectly reasonable for a model defining the meaning of the language.

Börger and Rosenzweig refine this abstract unification process to a concrete one in which they allocate cells the way the WAM does (in [6]). They exhibit a mapping F that maps concrete states to abstract states (p. 47); F maps cells in the concrete states in a one-to-one fashion to variables in the abstract states (using *mk_var*), and the variables in the mapped state must satisfy the original constraints.[4] Unfortunately, they cannot. Consider what happens during execution of a program with the following (duplicate) clauses.

```
p(X) :- ...., q(X).
p(X) :- ...., q(X).
```

Assume that there are no other clauses for p/1.[5] In their target machine, the first will have a choicepoint, but the second will not, so the (stack) cell chosen to represent variable X may be different. If so, the proof map F will map these two

[4] F is in fact not a function, because the function *subst* that it refers to is not welldefined. Since stack cells may be reused, the definition of *subst* would have to be changed so that later bindings override earlier ones.

[5] This counterexample works if other clauses for p/1 exist, so long as the last clause is one of the duplicates.

configurations to abstract states that have different variants. In order for this proof to work we must exhibit a *rename* function that agrees with the F mapping and also satisfies the original constraints. This function is only passed a clause and variable-renaming index; the clauses are obviously the same, and the index is as well, because the states used to evaluate them are both initialized with the same value of *vindex*. Therefore, the function will return the same variant, so it cannot agree with the F mapping.

In their first papers there was both a global renaming index (*vindex*) and an index in the interpreter state (*vi(currstate)*). Börger and Rosenzweig simplify this in [7], which is a revised version of [5] and [6]. The global variable *vindex* is renamed *vi*, and the function *vi* is eliminated. The *rename* function is defined on p. 30, and it is used (via the priming convention also defined on p. 30) to rename the candidate clause on p. 33. Since the variant used now depends on a global variable, and that variable is unaffected by backtracking (pg. 31), the problem discussed above does not occur; every candidate clause that is used (successfully) for resolution gets a new index, so two occurrences of the same term are never forced to be the same.

However, the revised version is still incorrect, and it is in fact easier to see why. We must still assume that the terms produced by the *rename* function have disjoint variables, that is, every "call" to *rename* produces new variables. The refined interpreter does not do this, since it uses stack cells to represent terms and these cells are reused. As before, concrete states are mapped to abstract states (p. 73), and these abstract states must meet the original constraints, but they cannot, since stack cells are re-used but variables never are.

References

[1] Wolfgang Ahrendt. Von Prolog zur WAM: Verifikation der Prozedurübersetzung mit KIV. Master's thesis, Universität Karlsruhe, December 1995.

[2] Hassan Aït-Kaci. *Warren's Abstract Machine: a Tutorial Reconstruction.* MIT Press, 1991.

[3] E. Börger. A Logical Operational Semantics for Full Prolog. In *Mathematical Foundations of Computer Science 1990*, volume 452 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 1990.

[4] Saumya K. Debray and Prateek Mishra. Denotational and Operational Semantics for Prolog. *Journal of Logic Programming*, 5:61–91, 1988.

[5] D. Rosenzweig E. Börger. From Prolog Algebras Towards WAM—a Mathematical Study of Implementation. In *4th Workshop on Computer Science Logic*, volume 533 of *Lecture Notes in Computer Science*, pages 31–66. Springer-Verlag, 1991.

[6] D. Rosenzweig E. Börger. WAM Algebras—a Mathematical Study of Implementation, part 2. In *1st Russian Conference on Logic Programming*, volume 592 of *Lecture Notes in Computer Science*, pages 35–54. Springer-Verlag, 1991.

[7] D. Rosenzweig E. Börger. The WAM—Definition and Compiler Correctness. In *Logic Programming: Formal Methods and Practical Applications*, pages 21–90. Elseview Science B.V., 1995.

[8] Michael J.C. Gordon and T. F. Melham. *Introduction to HOL, a Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[9] David Gries. *The Science of Programming.* Springer Verlag, 1981.

[10] N. D. Jones and A. Mycroft. Stepwise Development of Operational and Denotational Semantics for Prolog. In *Int. Symp. on Logic Programming 2/84*, pages 289–298, 1984.

[11] P. Kursawe. How to Invent a Prolog Machine. In *3rd Int. Conf. on Logic Programming*, volume 225 of *Lecture Notes in Computer Science*, pages 134–148. Springer-Verlag, 1986.

[12] Leslie Lamport. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994.

[13] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[14] Ulf Nilsson. Towards a Methodology for the Design of Abstract Machines for Logic Programming Languages. *Journal of Logic Programming*, 16:163–189, 1993.

[15] Lawrence C. Paulson. *Isabelle: a Generic Theorem Prover.* Springer Verlag, 1994.

[16] Cornelia Pusch. Verification of Compiler Correctness for the WAM. In *9th Int. Conf. on Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*, pages 347–362. Springer-Verlag, 1996.

[17] Peter Van Roy. The Wonder Years of Sequential Prolog Implementation. *Journal of Logic Programming*, 19:385–441, 1994.

[18] David M. Russinoff. A Verified Prolog Compiler for the Warren Abstract Machine. *Journal of Logic Programming*, 13:367–412, 1992.

[19] Gerhard Shellhorn and Wolfgang Ahrendt. Verification of a Prolog Compiler - First Steps with KIV. Technical report, Universität Karlsruhe, 1996.

[20] G. Shellhorn W. Reif and K. Stenzel. Interactive Correctness Proofs for Software Modules using KIV. In *Tenth Annual Conference on Computer Assurence*, 1995.

[21] D.H.D. Warren. An Abstract Prolog Instruction Set. Technical report, SRI International, Menlo Park, California, 1983.