Mechanising Aspects of miniKanren in HOL

Ramana Kumar

A thesis submitted in partial fulfillment of the degree of
Bachelor of Philosophy (Honours) at
The Australian National University

May 2010

© Ramana Kumar

Typeset in Computer Modern by TEX and LATEX 2_ε.

Except where otherwise indicated, this thesis is my own original work.

Ramana Kumar
7 May 2010

Acknowledgements

The main person I want to thank is my supervisor, Michael Norrish. As a PhB student, I have had many supervisors, some good and some bad. So I knew how important it would be for me to have a good supervisor for Honours. I couldn't have done better than Michael. He was happy to meet me every week, to read my hundreds of email messages, to write a conference paper with me, and to support me through writing this thesis. I wish every student could be so lucky. I thank Kee-Siong Ng for recommending him to me. Michael is also largely responsible for the termination argument for the right-hand-side checking walk in Section 2.5.1.

I thank Daniel Friedman and William Byrd for introducing me to miniKanren, and indeed to research on programming languages. I am working in the right area now, and it's mainly because of the happy research environment they, along with Joseph Near, provided for me when I was on exchange. Will and Dan both also read drafts of this thesis and gave me helpful criticism.

I also thank Katja Grace for reading a draft and giving me her comments.

Abstract

Focussing on the central notion of unification, we mechanise aspects of the miniKanren logic programming language.

We consider unification of (traditional) first-order terms, and also of *nominal* terms. We verify *termination* and *correctness* for each algorithm. The algorithms are written in an *accumulator-passing* style: taking a substitution as input, and returning an extension of that substitution on success. This style of algorithm is more difficult to reason about than the usual "direct style" of recursion. Both algorithms use *triangular substitutions*, which are important for performance, and have not been treated before. Indeed, a necessary preliminary is the development of a theory of triangular substitutions.

Subsequently, we provide three formal, machine-checked definitions of the semantics of miniKanren. We prove a *soundness* result relating the denotational and operational semantics, and also generate an experimentally-validated *executable* semantics.

All of this work was performed in the HOL4 proof assistant.

Contents

Acknowledgements **v**

Abstract **vii**

1 Introduction **1**

1.1 Overview . . . 1
1.2 Contributions . . . 3
1.3 Logic programming with miniKanren . . . 3
1.4 Theorem proving with HOL4 . . . 7
1.4.1 Basic concepts and notation . . . 8
Types and terms . . . 8
Definitions . . . 8
Theorems and theories . . . 10
1.4.2 Existing theories in HOL4 . . . 11
Sets, bags, lists . . . 11
Pairs, options, numbers . . . 11
Finite maps . . . 12
Relations . . . 12
1.4.3 Example proving sessions . . . 12

2 Triangular Substitutions **19**

2.1 Introduction . . . 19
2.1.1 Terms . . . 19
2.1.2 Substitution . . . 20
2.1.3 Composition and triangular form . . . 22
Etymology . . . 23
2.1.4 Persistence and search trees . . . 24
2.2 Well-formedness . . . 26
2.3 Substitution application . . . 28
2.3.1 `walk` . . . 29
2.3.2 Termination of `walk*` . . . 29
2.4 Collapsing triangular substitutions . . . 30
2.5 Association lists . . . 31
2.5.1 The right-hand-side check . . . 33

3 First-Order Unification **35**
3.1 Introduction . . . 35
3.2 The occurs check . . . 36
3.3 Definition . . . 37
3.4 Termination . . . 38
3.5 Correctness . . . 40
3.5.1 Soundness . . . 41
3.5.2 Generality . . . 41
3.5.3 Completeness . . . 42
3.6 Other algorithms . . . 42

4 Nominal Unification **47**
4.1 Introduction . . . 47
4.2 Nominal terms . . . 49
4.3 Definition . . . 51
Termination . . . 53
4.4 Correctness . . . 53
4.4.1 Soundness . . . 55
4.4.2 Generality . . . 56
4.4.3 Completeness . . . 57
4.5 Benchmarks . . . 58

5 miniKanren Semantics **61**
5.1 Introduction . . . 61
5.2 Syntax . . . 62
Well-formed programs . . . 63
5.3 Logical semantics . . . 65
5.4 Operational semantics . . . 67
5.4.1 Structured bags . . . 68
5.4.2 The state transition relation . . . 69
5.4.3 Lazy lists . . . 72
5.5 Soundness . . . 73
5.6 Executable semantics . . . 76

6 Conclusion **81**
6.1 The import of this work . . . 81
6.2 Related work . . . 81
6.2.1 First-order unification . . . 81
6.2.2 Nominal unification . . . 82
6.2.3 Semantics of logic programming languages . . . 82
6.3 Future work . . . 83

A List of Theorems in Mechanised Theories **85**
A.1 term . . . 85
A.2 subst . . . 86
A.3 collapse . . . 89
A.4 walk . . . 90
A.5 walkstar . . . 92
A.6 unifDef . . . 93
A.7 unifProps . . . 96
A.8 nterm . . . 96
A.9 nsubst . . . 97
A.10 dis_set . . . 97
A.11 apply_pi . . . 97
A.12 nwalk . . . 98
A.13 nwalkstar . . . 98
A.14 nunifDef . . . 98
A.15 nunifProps . . . 98
A.16 mKSyntax . . . 108
A.17 sbag . . . 109
A.18 mKAbs . . . 109
A.19 mKSoundness . . . 112
A.20 mKImp . . . 112

Bibliography **113**

Chapter 1

Introduction

1.1 Overview

My thesis is that the miniKanren approach to logic programming—primarily, the use of triangular substitutions—has a good theoretical foundation. I show this by proving the algorithms that work on substitutions in miniKanren correct, and by giving the system formal semantics, in the interactive theorem prover HOL4. The rest of this section is a brief overview of logic programming to put miniKanren in context.

A recurring theme in computer science is the goal of writing programs that are correct (especially, correct by construction). A computer program is correct if it does what it's supposed to do, according to some specification of its desired behaviour. Correctness means three layers must link up properly. The program's behaviour (the actions taken by the computer, like displaying an answer on the screen) should correspond to the program text (the source code). The text of the program should conform to the specification. And finally, the specification should be an accurate description of what we want the program to do.

Logic programming (LP) is a way to use the same piece of text for both the specification and the source code, thereby eliminating disagreement between them. The source code of a logic program corresponds directly to a formula (or many formulas comprising a theory) in some logic. This formula describes what should hold of solutions to a problem. It is then up to the logic programming system to read the formula and compute the solutions.

Formulas in LP usually contain (at least) equalities between terms, existentially quantified variables, conjunctions, and disjunctions. For example, take the formula

$$\exists x\, y\, z.\, x = \langle y,\, z\rangle \land (y = 3 \lor z = 3)$$

This says there exist values x, y, and z such that x is the pair of y and z, and one of y or z is equal to 3. It can be interpreted as a logic program where the goal might be to find all the possible values for x.

A logic programming system uses two components to find solutions: unification, and search. (We will illustrate these concepts in Section 1.3, and explain unification further in Chapter 3 and search further in Section 2.1.4.) Unification is used on each term equality problem to compute a local solution (a unifying substitution) or determine

that none exists. The search is used to find the collections of term equality problems that must simultaneously be solved. The search directs the sequence of unifications, and is directed by the success or failure of individual unifications.

Unification has been studied extensively, both for LP and for other applications including type inference and automated theorem proving. Some applications, LP in particular, call for sequences of related unification problems to be solved and their results composed. Traditionally, LP systems use a unification algorithm equally suited to a one-shot problem. The LP system miniKanren [Byrd 2009; Friedman et al. 2005], in contrast, uses a unification algorithm written in an accumulator-passing style that is better suited to its role as a repeated component of a larger search for solutions.

One of the first and most popular LP languages is Prolog [Kowalski 1974], and many developments in the field have built upon Prolog. miniKanren was designed 30 years after Prolog for teaching and research of pure logic programming (so-called "relational" programming). Considering only the problems to which it's suited and the approach taken to solving them, miniKanren is essentially the same as (pure) Prolog. However, there are at least three interesting aspects of miniKanren that have sprung from its fresh design and haven't received much attention in LP's 50 year history:

1. An accumulator-passing unification algorithm, using triangular substitutions (already mentioned);

2. Lexical scoping, as found in functional languages like Scheme and ML, in contrast to the flat list of definitions in Prolog programs; and

3. An interleaving streams-based approach to search, which offers a complete search strategy (in contrast to Prolog's incomplete depth-first search) that is more efficient than breadth-first search. In this connection, miniKanren's evaluation strategy is not obviously equivalent to Prolog's standard SLD resolution (although it may in the future be shown equivalent).

My Honours project is a study of miniKanren, aimed at giving the system a better theoretical foundation. My main focus is on the unification algorithms in miniKanren and its nominal logic counterpart αKanren (these come under item 1 above). The rest of my work, described in Chapter 5, is on the semantics of miniKanren programs, which are influenced by all three items on the list.

All the results in this thesis were formulated and verified in the interactive theorem prover HOL4. Formal systems, *i.e.*, the mathematical tools we use to express our theorems and their proofs, call out for computer implementation. A formal system has a precisely defined language and rigid rules for constructing and manipulating proofs. Keeping track of the details is the perfect job for a computer. As explained by the authors of the POPLmark challenge [Aydemir et al. 2005], proofs about programming languages are especially apt for mechanisation, because they are usually large and tedious.

1.2 Contributions

The main payload of my work this year is the HOL theories about triangular substitutions, unification, and miniKanren. The thesis itself is a standalone guide to the results proved in these theories. A summary of the main results is as follows.

- A well-formedness condition on triangular substitutions, characterising those substitutions with fixpoints (Section 2.2);
- A proof of the termination of an algorithm, `walk*`, for applying triangular substitutions (Section 2.3);
- Termination proofs for recursive descent unification algorithms written in accumulator-passing style for first-order (`unify`, Sections 3.3 and 3.4) and nominal (`nomunify`, Section 4.3) terms;
- Correctness of `unify` and `nomunify`: well-formedness preservation, soundness, completeness, and generality (Sections 3.5 and 4.4); and
- Soundness of an operational semantics for miniKanren with respect to a high-level logical semantics (Section 5.5).

1.3 Logic programming with miniKanren

In this section we will get a taste of logic programming by looking at a few small examples. The main purpose is to understand the roles of unification and search in a logic programming system. We begin with a simple miniKanren program and its result (Listing 1.1). This program corresponds to the formula

$$\exists q\, x\, y.\, x = 3 \land y = 4 \land q = \langle x,\, y\rangle$$

and the goal to find all the possible values for q. The last line (starting with $\Rightarrow$) is the list of answers computed by miniKanren, containing the sole answer $\langle 3,\, 4\rangle$.

miniKanren is implemented as an embedding in the Scheme programming language [Sperber et al. 2009]. There are four miniKanren operators, which are added to Scheme's constructs for binding variables and creating and calling procedures. We can see three of them in Listing 1.1: **`run*`**, **`exist`**, and $\equiv$. The first one, **`run*`**, encloses

```
(run* (q)
  (exist (x y)
    (≡ x 3)
    (≡ y 4)
    (≡ q ⟨x, y⟩)))
⇒ (⟨3, 4⟩)
```

Listing 1.1: A miniKanren program.

the rest of the program, and indicates the *query variable*, in this case q, whose possible values are to be found. The **exist** operator introduces new existential variables, in this case x and y, which can be referred to in the body of the **exist** form. The $\equiv$ operator specifies that its two arguments should be made equal by binding existential variables as necessary. The process of determining whether and how terms can be made equal is called *unification*, thus $\equiv$ is a call on the system's unification procedure.

Unification fails when the terms cannot be made equal. If we add another line to the program above, asking that $\langle x, y\rangle = \langle y, x\rangle$, then miniKanren returns an empty list of answers, indicating that there are no possible answers (Listing 1.2). The unification on line 3 below is not necessarily the one that fails. The order in which miniKanren makes the calls is not specified, but if we assume they are made in the order they appear in the program, then the failure will happen at line 5. Trying to make y equal 4 fails because a consequence of lines 3 and 4 is that $x = y = 3$.

```
1 (run* (q)
2   (exist (x y)
3     (≡ ⟨x, y⟩ ⟨y, x⟩)
4     (≡ x 3)
5     (≡ y 4)
6     (≡ q ⟨x, y⟩)
7 ⇒ ()
```

Listing 1.2: A conjunct added to Listing 1.1 leading to failure.

The last miniKanren operator is **conde**, which is used to write a disjunction of conjunctions. We have already seen conjunction in the bodies of **exist** and of **run***: the subexpressions in each of those forms were conjoined together. The next program (Listing 1.3) adds disjunction, using **conde**. The first disjunct is $q = 3 \wedge 3 = q$, which

```
(run* (q)
  (conde
    ((≡ q 3) (≡ 3 q))
    ((≡ 3 q))
    ((≡ q 4))
    ((≡ 3 3))))
⇒ (3 3 4 _0)
```

Listing 1.3: A program illustrating **conde**.

is true if q is bound to 3, so this disjunct contributes one 3 to the answer list. The second disjunct, $3 = q$, contributes the second 3. The third disjunct, $q = 4$, contributes the 4. The last disjunct, $3 = 3$, is true no matter the value of q. The answer $__0$ means that the query variable can be bound to anything.

As we have seen, a formula with disjunction may yield more than one answer. The answers are computed by trying each of the disjuncts in turn, collecting answers from

each one. This is the *search* component of logic programming. In fact, miniKanren uses an *interleaving* search to find answers, effectively running all the disjuncts in parallel. With or without interleaving, however, disjuncts should not interfere. In line 4 of Listing 1.3, q is made equal to 3, but in line 5 the same variable q is made equal to 4. This is possible because q is unbound before each disjunct is examined, and the bindings inferred from one disjunct are kept separate from the other disjunct.

```
1  (run* (q)
2    (conde
3      ((≡ q 1) (≡ 1 0))
4      ((exist (x y)
5         (conde
6           ((≡ x 3) (≡ y 4))
7           ((≡ q ⟨x, y⟩)))))
8      ((exist (x y)
9         (≡ ⟨x, y⟩ ⟨y, x⟩)
10        (≡ q ⟨x, y⟩)))))
11 ⇒ (⟨_0, _0⟩ _0 ⟨_0, _1⟩)
```

Listing 1.4: A more involved program.

Listing 1.4 combines everything we've seen so far. The corresponding formula in logic is

$$\begin{aligned}&\exists q.\, (q = 1 \land 1 = 0) \lor\\&\quad(\exists x\, y.\, (x = 3 \land y = 4) \lor q = \langle x,\, y\rangle) \lor\\&\quad(\exists x\, y.\, \langle x,\, y\rangle = \langle y,\, x\rangle \land q = \langle x,\, y\rangle)\end{aligned}$$

The outer **conde** (line 2) contains three disjuncts. The first disjunct (line 3) contains a contradictory conjunct, $1 = 0$, so it contributes no answers to the final list. The second disjunct (lines 4–7) introduces two existential variables whose scope is this disjunct only, and contains a nested disjunction. Line 6 doesn't contain a contradiction, and doesn't say anything about q, so it contributes the answer $__0$. Line 7 says that q must be the pair $\langle x,\, y\rangle$, but doesn't say anything else about x and y. The answer is $\langle __0, __1\rangle$ indicating that q can be any pair, where the sides of the pair are not necessarily equal. The final disjunct (lines 8–10) introduces two fresh existential variables and has two unifications in conjunction. Line 10 binds q to $\langle x,\, y\rangle$, and a consequence of line 9 is that $x = y$. Therefore the answer is $\langle __0, __0\rangle$ indicating a pair whose sides must be the same, arbitrary value.

The last feature of miniKanren we haven't yet covered is the definition of new possibly recursive relations. Relations are sometimes called predicates in the context of first-order LP. We will describe them by example. The example we've already seen is the one built in to miniKanren, the $\equiv$ relation, which holds when its arguments can be made equal. Other examples include the addition relation, which holds of (x, y, z) when $x + y = z$, and the append relation which holds of (l_1, l_2, l_3) when l_3 is the list

concatenation of l_1 and l_2.

From a programming perspective, a relation is a procedure that takes terms as arguments and either succeeds or fails. A single call to a relation may succeed multiple times if there are multiple possible bindings for the variables in the arguments. Thus the less-than relation on natural numbers, which holds of (x, y) when $x < y$, would succeed twice when applied to arguments $(q, 2)$ by binding q to 0 in one case, and to 1 in the other. Relations like less-than and addition are not built into miniKanren, but [Kiselyov et al. 2008] show how to define them, and make a good case that terminating relational arithmetic over unbounded natural numbers is a subtle problem.

```
(define appendo
  (λ (l s ls)
    (conde
      ((≡ l ()) (≡ ls s))
      ((exist (a d)
         (≡ l ⟨a, d⟩)
         (exist (ds)
           (≡ ls ⟨a, ds⟩)
           (appendo d s ds)))))))
```

Listing 1.5: Definition of appendo in miniKanren.

Relations are implemented as Scheme procedures in miniKanren, and are therefore created by Scheme's **λ** form. Listing 1.5 shows a definition of the append relation, as defined in [Byrd 2009]. (It is a miniKanren convention to end relation names, apart from ≡, with 'o'.) The definition is not a complete miniKanren program since it doesn't include a **run*** form, but it corresponds to a formula in logic that specifies the `appendo` constant recursively.

$$
\begin{aligned}
&\texttt{appendo}(l, s, ls) \iff \\
&\quad l = () \land ls = s \lor \\
&\quad \exists a\, d.\, l = \langle a,\, d\rangle \land \\
&\qquad \exists ds.\, ls = \langle a,\, ds\rangle \land \texttt{appendo}(d, s, ds)
\end{aligned}
$$

A list in Scheme is either () or a pair $\langle a,\, l\rangle$ where l is a list. The definition of `appendo` can be understood as follows. The list ls is the concatenation of l and s whenever l is empty and $ls = s$, or whenever there exist a head a and tail d such that $l = \langle a,\, d\rangle$, and an intermediate list ds such that d and s concatenate to form ds and ls is a added to the front of ds. The definition of the relation corresponds closely to the recursive definition of a function that would compute list concatenation. The correspondence between definitions of relations in logic programming and of functions in functional programming is explored in [Friedman et al. 2005].

We can use the newly defined relation in some programs with queries, as shown in Listing 1.6. The first run calculates the concatenation of two lists, just as the append

function in a functional language might. The second run shows how the same relation, in an LP system, can be run backwards to infer what the first list must be given the second list and their concatenation. The third run, which asks for the first list given only that the concatenation is $\langle 1, \langle 2, () \rangle \rangle$, yields multiple answers, and shows that computing in an LP system means doing a search over all possible answers, finding, in this case, all the prefixes of the output list. The disjunction responsible for the multiplicity of answers is in the definition of `appendo`, where one disjunct assumes the first list is empty, and the other assumes it isn't. In this run, the first and second lists are both initially unbound variables, so both assumptions are non-contradictory, and both disjuncts yield answers.

```
(run* (q) (appendo ⟨1, ()⟩ ⟨2, ⟨3, ()⟩⟩ q)
⇒ (⟨1, ⟨2, ⟨3, ()⟩⟩⟩)

(run* (q) (appendo q ⟨3, ()⟩ ⟨1, ⟨2, ⟨3, ()⟩⟩⟩))
⇒ (⟨1, ⟨2, ()⟩⟩)

(run* (q) (exist (x) (appendo q x ⟨1, ⟨2, ()⟩⟩)))
⇒ (() ⟨1, ()⟩ ⟨1, ⟨2, ()⟩⟩)
```

Listing 1.6: Programs using `appendo`.

We have seen that new relations can be defined in miniKanren as functions built out of conjunctions and disjunctions of other relations (ultimately $\equiv$). These building blocks are not enough to build programs to represent any formula in first-order logic. For example, there is no way to define a general relation $\not\equiv$ that relates two terms only when they are *not* equal. The programs that can be made correspond to a well-known class of formulas called Horn clauses [Horn 1951; Andréka and Németi 1980].

This thesis is mostly concerned with triangular substitutions and unification. Our focus will turn to miniKanren again in Chapter 5 where we develop semantics of miniKanren that depend on the unification algorithm in Chapter 3. In Section 5.2, we will define a simplified model of the miniKanren syntax seen in this section, removing the dependence on Scheme by isolating the forms that can be regarded as miniKanren proper.

1.4 Theorem proving with HOL4

HOL stands for higher order logic. A logic is a mathematical system for expressing precise statements together with rules for constructing proofs that certain statements are true. HOL is as an extension of first-order logic, the logic we saw in the previous sections, in the sense that any first-order formula can be written in HOL. Additionally, HOL allows variables that stand for functions, which makes it easy to define and reason about functions in HOL.

HOL is also the name of the computer system, the latest version of which is HOL4 [Norrish and Slind 1998], that implements the logic and facilitates the con-

struction of mechanically verified theories. HOL4 is an example of the *LCF approach* to theorem proving systems (described, *e.g.*, in [Gordon 2000]), wherein theorems are represented by an abstract data type in a programming language, and the only constructors of theorems are functions implementing the primitive inference rules of the logic. The implementation language (*i.e.*, metalanguage) for HOL4 is Standard ML (SML) [Milner et al. 1997], which is strongly typed and enforces abstraction boundaries with static type checking. Isolating theorem constructors in a small kernel allows users to develop high-level proof tools by programming in SML without compromising the security of the system: any value of the theorem type must always have come in some way from the kernel implementing the inference rules. Slind and Norrish provide introductions to HOL4, its history, and uses in [Norrish and Slind 2002] and [Slind and Norrish 2008].

1.4.1 Basic concepts and notation

Types and terms Every term in HOL has a type. Examples of types include `bool` (Booleans), `num` (natural numbers), and `string` (strings). Type operators build types out of type arguments, for example the type of total functions from numbers to Booleans is $\texttt{num} \rightarrow \texttt{bool}$ and is built with the $\rightarrow$ type operator. Usually type operators are written postfix: `num list` is the type of a list of numbers. Types can be polymorphic, for example lists whose elements are of type α have the type α `list`. Every type is inhabited; the constant `ARB` for each type denotes an arbitrary object of that type.

There is no special syntactic class of formulas in HOL, because terms of type `bool` are suitable. HOL syntax for Boolean terms uses standard connectives and quantifiers ($\wedge$, $\forall$ *etc.*). Of course $\wedge$, for example, is simply a function of type $\texttt{bool} \rightarrow \texttt{bool} \rightarrow \texttt{bool}$. The constants true and false are written `T` and `F`.

Function application is usually written in prefix notation without brackets, for example $f\ \ a_1\ \ a_2$ instead of $f(a_1, a_2)$, but there are exceptions like $(\wedge)$, which is infix. We use parentheses to write the name of an infix function, *i.e.*, as a term by itself without any arguments. Functions in HOL are curried, meaning they only take a single argument, but may return a function that will consume the next argument. A single argument, however, may be of a tuple type such as $\alpha\ \#\ \beta$, the type of pairs. Examples: in $f\ \ a_1\ \ a_2$, f has type $\alpha \rightarrow \beta \rightarrow \gamma$, and in $g\ \ (a_1, a_2)$, g has type $\alpha\ \#\ \beta \rightarrow \gamma$.

Lambda expressions denote functions, for example $\lambda x\ y.\ x + y + 1$ has type $\texttt{num} \rightarrow \texttt{num} \rightarrow \texttt{num}$ and is the function that returns one more than the sum of two numbers. Applying this function to a single argument, 3, would yield the function $\lambda y.\ 3 + y + 1$. Let expressions are defined as the application of a lambda expression, for example $\textsf{let}\ x = f\ \ y\ \textsf{in}\ x + 1$ means $(\lambda x.\ x + 1)\ (f\ \ y)$.

Definitions HOL allows the definition of new constants, and we use this facility to define functions. Since all functions in HOL are total, newly defined functions must have been proven to terminate on all inputs. For primitive recursive functions (such as Definition 2 on page 20), and some simple other cases, the termination proof is automatic. When the termination proof is not automatic, HOL4 generates termination

conditions that, if they can be proved manually, are sufficient to make the definition. We will see this process in Section 2.3 for `walk*`, and in Section 3.4 when we are defining `unify`.

Defining a function amounts to providing a name (a new constant), proving a characterising theorem, and, in the case of non-primitive recursive functions, proving an induction theorem. The characterising theorem may be as simple as

$$\vdash \texttt{ADD2}\ x = x\ \texttt{+ 2}$$

or may be the conjunction of various recursive clauses, such as

$$\texttt{FACT 0} = \texttt{1}$$
$$\texttt{FACT (SUC}\ n\texttt{)} = \texttt{SUC}\ n\ \texttt{*}\ \texttt{FACT}\ n$$

While proving some goal about the new constant, these clauses can be used to rewrite a call of the function, *e.g.* `FACT (SUC 0)`, to a simpler equivalent: `SUC 0 * FACT 0`.

The induction theorem can be used to prove statements by so-called recursion induction, following the shape of the recursion of the newly defined function. Since the function terminates, any proposition that holds of the original arguments whenever it holds for the arguments of all recursive calls must hold for all possible arguments. The induction theorem captures this reasoning, and is especially useful for proving theorems about the function itself. In the case of primitive recursive functions, no induction theorem is necessary because there will be an induction theorem for the type of the argument that will do the job. The technology that proves the characterising and induction theorems, starting from recursive clauses, or generates termination proof conditions in case termination can't automatically be proved, is a library called TFL, and is described in [Slind 1999].

HOL4 also comes with a facility for defining inductive relations. An inductive relation is defined by a list of possibly recursive rules, where each rule is an implication with an instance of the relation on the right-hand-side. We write these rules with the conclusion below a horizontal line, and the hypotheses above. The rules from Definition 19 (in Section 3.2) are repeated below.

$$\frac{v \in \texttt{vars}\ t \quad v \notin \texttt{FDOM}\ s}{\texttt{oc}\ s\ t\ v} \qquad \frac{u \in \texttt{vars}\ t \quad \texttt{vwalk}\ s\ u = t' \quad \texttt{oc}\ s\ t'\ v}{\texttt{oc}\ s\ t\ v}$$

The relation is taken to be the smallest relation that makes the rules true. After defining an inductive relation, HOL4 automatically proves an induction theorem and a cases theorem for the relation. The induction theorem is suitable for proving a statement of the form "If the original relation holds of some arguments, then some other property also holds of the same arguments". As an example, for `oc` we have the following induction theorem that will work for any property oc' satisfying the same rules as `oc`.

$$\begin{array}{l}\vdash (\forall t\ v.\ v \in \texttt{vars}\ t \wedge v \notin \texttt{FDOM}\ s \Rightarrow oc'\ t\ v) \wedge \\ \quad (\forall t\ v\ u\ t'.\ u \in \texttt{vars}\ t \wedge \texttt{vwalk}\ s\ u = t' \wedge oc'\ t'\ v \Rightarrow oc'\ t\ v) \Rightarrow \\ \quad\quad \forall a_0\ a_1.\ \texttt{oc}\ s\ a_0\ a_1 \Rightarrow oc'\ a_0\ a_1\end{array}$$

The cases theorem enumerates the rules that could have generated an instance of a relation. The example for oc is below. It is called a cases theorem (not to be confused with case expressions) because it enables case analysis of any instance $\texttt{oc}\ s\ t\ v$ of the relation.

$$\begin{array}{l}\vdash \texttt{oc}\ s\ t\ v \iff \\ \quad v \in \texttt{vars}\ t \land v \notin \texttt{FDOM}\ s \lor \\ \quad \exists u\ t'.\ u \in \texttt{vars}\ t \land \texttt{vwalk}\ s\ u = t' \land \texttt{oc}\ s\ t'\ v\end{array}$$

We have looked at defining term constants. We can also define new types. Abbreviations can be introduced for existing types, for example `string` actually abbreviates `char list`. Entirely new types can be defined inductively, in a similar way to data types in ML, by declaring the name and constructors of the new type. For example, the type of miniKanren terms in Definition 1, repeated below, is a new inductive data type with three constructors.

α `term = Var of num | Pair of` α `term =>` α `term | Const of` α

The constructors are simply functions into the new type; for example `Var` has type `num` $\rightarrow$ α `term`. The term type is recursive, since the `Pair` constructor takes terms as arguments. It is also polymorphic, and depends on the type α of constants. Terms with numbers representing constants would have type `num term`. Values with inductively defined types can be deconstructed by case expressions, as explained in the paragraph on pairs below. As an example, the expression case `Var 1` of `Var` $n \rightarrow n \parallel$ _ $\rightarrow$ `0` is equal to `1`.

Theorems and theories A theorem in HOL4 contains a term of type `bool` (the conclusion) and a list of assumptions (all our theorems will have the empty list). Theorems are values with SML type `thm`, which means they are produced by the HOL4 kernel. Thus all theorems are valid (assuming the kernel implements the inference rules of HOL). We write theorems with a turnstile, for example here's a theorem asserting extensional equality of functions

$$\vdash \forall f\ g.\ f = g \iff \forall x.\ f\ x = g\ x$$

Theorems can contain free variables; derived inference rules include instantiation of free variables, generalisation, and specialisation. We usually write theorems with the outermost universally quantified variables all specialised, to save space, as in

$$\vdash f = g \iff \forall x.\ f\ x = g\ x$$

Function definitions, like of `FACT` shown earlier, are also theorems, although we print them without the turnstile and without the '$\land$'s separating the clauses.

A theory is a collection of theorems together with a signature specifying the term and type constants. Theories are implemented as SML structures. One theory can inherit signature information from other theories, which become its ancestors in a theory dependency graph. Our theories about substitutions, substitution application, unification, *etc.*, will build on some of the theories provided in the HOL4 distribution, which are described in the next section.

1.4.2 Existing theories in HOL4

Sets, bags, lists Sets in HOL are identified with their characteristic functions. Thus the type α `set` is an abbreviation for $\alpha \to$ `bool`. Common notation for sets, such as $x \in s$, `FINITE` s, and $s_1 \cup s_2$, have the usual meanings. The empty set is written $\{\,\}$. x `INSERT` s means the set s with element x inserted, *i.e.*, $s \cup \{x\}$.

Bags, also known as multi-sets, are like sets except they keep track of the number of copies of each element they contain. The type α `bag` is an abbreviation for $\alpha \to$ `num`. Bag union and difference, which amount to element-wise addition and subtraction, are written $b_1 \uplus b_2$ and $b_1 - b_2$. For example

$$\{|1;\ 1;\ 2|\} - \{|2;\ 3|\} \uplus \{|1;\ 3|\} = \{|1;\ 1;\ 1;\ 3|\}$$

There is a `3` in the result because subtracting an element from a bag that doesn't contain the element does nothing. More generally, natural number subtraction obeys $m - n = 0$ when $n \geq m$.

Lists are an inductive type with the empty list constructor `[]`, and the polymorphic constructor `CONS`. The term x`::`ls denotes `CONS` applied to two arguments, and means the element x added onto the front of the list ls. The concatenation (*i.e.*, append) of two lists is written l_1 `++` l_2. A concrete list can be written in list syntax `[1; 2; 3]`, although usually we just deal with variables of a list type like l_1 in the previous sentence. The length of a list is written `LENGTH` ls. The term `MAP` f ls denotes the list obtained by replacing every element x in ls by f x. The term `EVERY` P ls is true exactly when P x is true for every element x in the list. The term `ALL_DISTINCT` ls is true exactly when the elements of ls are all distinct. The set of elements in a list is written `set` ls.

Pairs, options, numbers Pairs are written `(`a`,`b`)` and the type constructor is α # β. Tuples are built by nesting pairs. Tuples and inductive data types can be deconstructed by case expressions. The notation is

$$\mathsf{case}\ t\ \mathsf{of}\ (x,3) \to f\ x \parallel (2,y) \to g\ y$$

Patterns may include underscores as wildcards. We can also extract the halves of a pair directly with the functions `FST` and `SND`, *e.g.* `FST (`a`,`b`)` $= a$.

The type `(:`α `option)` is an option type, and can take the values `NONE` or `SOME` x where x is of type α. The `do` notation is used for writing in monadic style. We only use it to express bind in the option monad: the term `do` $y \leftarrow f\ x$`;` $g\ y$ `od` means `NONE` if f x returns `NONE`. Otherwise, if f x returns `SOME` y, then the term is the result of applying g to y (giving a value of option type). The function `THE` returns the value in an option, and is defined as follows: `THE (SOME` x`)` $= x$ and `THE NONE = ARB`.

We will only use natural numbers in our theories, although HOL4 has theories of other numbers. The type `num` is inductive, with constructors `0` and `SUC`. Terms such as $n < m$ and n `*` m have the usual meanings (n is less than m, n times m). Iterated application of a function is written $f^n\ x$, meaning $f(f(\ldots f(x)))$.

Finite maps A finite map is like a function, but has an explicit, finite domain. The type is written $\alpha \mapsto \beta$, where α is the type of the keys (domain) and β the values (range). The domain of a finite map is written FDOM fm. The term fm ' k means application of a finite map, returning SOME v when the key k is in the domain, otherwise NONE. The finite map with an empty domain is written FEMPTY. The update of a finite map with a new key-value pair is written fm |+ (k,v). A sequence of updates from a list of key-value pairs is written fm |++ ls. The submap relation is written $fm_1 \sqsubseteq fm_2$, and means $\forall\, k\ v.\ fm_1$ ' k = SOME $v \Rightarrow fm_2$ ' k = SOME v. Composition of a function after a finite map is written $f \circ fm$. The function FUN_FMAP turns a function into a finite map by restricting its domain: FUN_FMAP f X denotes the finite map with domain X and range $f\ x$ for $x \in X$.

Relations Relations are represented as curried functions to bool. A binary relation on some type α is thus a function of type $\alpha \rightarrow \alpha \rightarrow$ bool. R^+ denotes the transitive closure (one or more steps) of a binary relation, and R^* the reflexive transitive closure (zero or more). NRC R n is the relation corresponding to exactly n steps of R, and is thus a subrelation of R^*.

The relation measure f, where f is of type $\alpha \rightarrow$ num, relates x and y if $f(x) < f(y)$. Thus measure f itself has type $\alpha \rightarrow \alpha \rightarrow$ bool. The lexicographic combination of two relations, R_1 LEX R_2, is the relation on pairs that holds between (a, b) and (c, d) if $R_1\ a\ c$ or else $a = c$ and $R_2\ b\ d$. Thus if R_1 has type $\alpha \rightarrow \alpha \rightarrow$ bool and R_2 has type $\beta \rightarrow \beta \rightarrow$ bool then R_1 LEX R_2 has type $\alpha \ \#\ \beta \rightarrow \alpha\ \#\ \beta \rightarrow$ bool. The relation inv_image R f, where f has type $\alpha \rightarrow \beta$ and R has type $\beta \rightarrow \beta \rightarrow$ bool, relates x and y if R (f x) (f y). Thus inv_image R f has type $\alpha \rightarrow \alpha \rightarrow$ bool. These three kinds of relation can work well together, for example

```
inv_image (measure pair_count LEX (⊆)) (λt. (t,vars t))
```

is a relation that relates two terms if either the first term has fewer pairs in it than the second, or else they have the same number of pairs and the variables of the first are a subset of the variables of the second. (pair_count is defined in Section 2.1.1.)

The one-step multi-set ordering relation for R holds between two finite bags b_1 and b_2, written mlt1 R b_1 b_2, if b_1 can be obtained by replacing an element e in b_2 by a finite bag of elements such that each element in the replacement is R-less than e. The transitive closure of mlt1 R is written mlt R. As an example, we have

```
mlt (<) {|1; 1; 2; 4|} {|2; 3; 4|}
```

because we can, for example, replace the 3 by the bag of smaller numbers {|1; 2|} and replace the 2 by {|1|} to obtain the bag on the left. Finally, a binary relation is well-founded, written WF R, if there is no descending infinite sequence of values with each pair related by R.

1.4.3 Example proving sessions

The usual method for proving theorems, and thereby building theories, in HOL4 is by applying *tactics* to *goals*. In this section we will see what those terms mean by looking

at a couple of examples of proof building. The examples also illustrate some of the conveniences and difficulties of working with an interactive theorem prover.

The first theorem we will look at proving is that `vars` t is a finite set. The function `vars`, defined in Definition 2 (page 20), computes the set of variables occurring in a miniKanren term (in our model of first-order terms, defined in Definition 1). We will prove the theorem by induction on terms, since they are an inductively defined data type.

To start proving the theorem, while running a HOL4 session, we set up an initial goal consisting of the statement of the theorem

```
FINITE (vars t)
```

and the empty list of assumptions. A goal is a pair of a statement and an assumption list, and during a single proof we may end up with many goals. To work on a goal, we apply tactics. In this case, the first tactic we apply is

```
Induct_on `t`
```

The tactic language is not fixed: users are able to write new tactics by defining an ML function. HOL4 manages the combination of individual tactics automatically, keeping track of the goal we're working on and of the overall proof. Applying the `Induct_on` tactic results in three subgoals, corresponding to the three constructors for α `terms`. The second subgoal has two assumptions, in a numbered list below the goal statement, which are the inductive hypotheses.

```
∀ n. FINITE (vars (Var n))

FINITE (vars ⟨t, t′⟩)
------------------------------------
  0. FINITE (vars t)
  1. FINITE (vars t′)

∀ a. FINITE (vars (Const a))
```

Now for each goal we can rewrite with the definition of `vars`. Usually we must work on one subgoal at a time, but we'll list them here simultaneously after the rewriting step.

```
∀ n. FINITE {n}

FINITE (vars t ∪ vars t′)
------------------------------------
  0. FINITE (vars t)
  1. FINITE (vars t′)

FINITE { }
```

The ML function `THEN`, called a tactical, combines two tactics into one that has the effect of the first tactic followed by the second. The combined tactic distributes the second tactic over all the subgoals generated by the first, so we could have reached the current proof state by starting at the initial goal and applying

```
Induct_on `t` THEN REWRITE_TAC [vars_def]
```

The remaining goals can all be solved by rewriting with various theorems about finite sets that come with HOL4's theory of sets. The tactical `THENL` is like `THEN` but allows us to give a separate tactic for each subgoal. So we could write a tactic to solve the initial goal as follows

```
Induct_on `t` THEN REWRITE_TAC [vars_def] THENL [
  REWRITE_TAC [FINITE_INSERT,FINITE_EMPTY],
  ASM_REWRITE_TAC [FINITE_UNION],
  REWRITE_TAC [FINITE_EMPTY]
]
```

`ASM_REWRITE_TAC` is a version of `REWRITE_TAC` that also looks at the assumptions. Rewriting the second subgoal with `FINITE_UNION` would produce the goal

```
FINITE (vars t) ∧ FINITE (vars t′)
------------------------------------
  0. FINITE (vars t)
  1. FINITE (vars t′)
```

but by looking at the assumptions, it can be solved at once.

This proof of `FINITE (vars` t`)` is in fact much longer than it needs to be. There is a first-order resolution theorem prover tactic, `METIS_TAC`, that can automatically make the simple reasoning steps we made above. A tactic solving the initial goal using `METIS_TAC`, which takes a list of theorems to use during proof search, is

```
Induct_on `t` THEN
METIS_TAC [vars_def,FINITE_INSERT,FINITE_EMPTY,FINITE_UNION]
```

Even better, in many cases, than a first-order proof search, is the use of repeated contextual rewriting. A powerful tactic, the stateful rewriting tactic `SRW_TAC`, can also solve our goal after we set up the term induction. The theorems about finite sets are automatic rewrites, so don't need to be passed explicitly to `SRW_TAC`. We can also make `vars_def` an automatic rewrite, so the complete tactic would be

```
Induct_on `t` THEN SRW_TAC [][]
```

"By induction, then rewriting" is a short proof. Although our initial goal of proving that the set of variables in a term is finite was very modest, this last tactic shows the power of tactic-based proof in HOL4 at its best. It also shows that all the details of a proof may not be readily apparent from the code for the tactic.

Our second example doesn't succumb to rewriting immediately, and better illustrates the idea that a proof in a theorem prover is often one or two levels more explicit than a non-mechanised proof. We will interactively build the tactic to solve Lemma 25, whose statement is repeated below.

$\vdash$ oc s t v $\wedge$ ($\forall w.$ $t \neq$ Var w) $\wedge$ wfs s $\wedge$ wfs s_2 $\Rightarrow$
$s_2 \lhd$ Var $v \neq s_2 \lhd s \lhd t$

In words: if, in the context of a substitution s, a variable v occurs in a non-variable term t, then applying another substitution s_2 to v will be different from applying s_2 to s applied to t. By "applying" here we mean repeated application until a fixpoint is reached, as described in Chapter 2. The notation $s \lhd t$ means applying s to t in this sense. We assume both substitutions are well-formed (`wfs`), which means the fixpoints exist. The statement is true because v is a proper subterm of $s \lhd t$. Whatever v is sent to by s_2 should therefore be a proper subterm of $s_2 \lhd s \lhd t$.

Talk about subterms suggests using term induction. We won't use term induction directly; instead we will use the following lemma which is proved by term induction.

Lemma. (`vars_measure`) *If v occurs in t then $s \lhd$ `Var` v is smaller than $s \lhd t$*

$\vdash$ $v \in$ vars t $\wedge$ ($\forall w.$ $t \neq$ Var w) $\wedge$ wfs s $\Rightarrow$
measure (pair_count $\circ$ walk* s) (Var v) t

To prove Lemma 25, we begin with `STRIP_TAC`, which moves the antecedent of the statement into the assumptions, giving us the goal

```
s2 ◁ Var v ≠ s2 ◁ s ◁ t
------------------------------------
0. oc s t v
1. ∀w. t ≠ Var w
2. wfs s
3. wfs s2
```

We deduce $v \in$ `vars` ($s \lhd t$) from assumption 0, by applying `METIS_TAC` with Lemma 18 (on page 37). We also deduce $\forall w.\ s \lhd t \neq$ `Var` w from assumption 1 by doing a case analysis on t then simplifying. So far, our tactic looks like

```
STRIP_TAC THEN
`v IN vars (walk* s t)`
  by METIS_TAC [oc_eq_vars_walkstar,IN_DEF] THEN
`!w. (walk* s t) <> Var w`
  by (Cases_on `t` THEN FULL_SIMP_TAC (srw_ss()) [])
```

We've written the tactic as it might be typed. To clarify, `walk* s t` means $s \lhd t$, and `oc_eq_vars_walkstar` is an SML variable bound to Lemma 18. The current goal is

```
s2 ◁ Var v ≠ s2 ◁ s ◁ t
------------------------------------
0. oc s t v
1. ∀w. t ≠ Var w
2. wfs s
3. wfs s2
4. v ∈ vars (s ◁ t)
5. ∀w. s ◁ t ≠ Var w
```

The next tactic we'll use is `IMP_RES_TAC`, which tries to generate consequences of a theorem using the assumptions in the goal. We use it on `vars_measure` lemma. Assumption 6 below, which says $s_2 \lhd$ `Var` v is smaller than $s_2 \lhd s \lhd t$, is the one we want, but `IMP_RES_TAC` also adds three others.

```
...
6. measure (pair_count ∘ walk* s₂) (Var v) (s ◁ t)
7. measure (pair_count ∘ walk* s) (Var v) (s ◁ t)
8. ∀v. v ∈ vars t ⇒ measure (pair_count ∘ walk* s₂) (Var v) t
9. ∀v. v ∈ vars t ⇒ measure (pair_count ∘ walk* s) (Var v) t
```

The last few steps of the proof start by assuming the goal is false ("proof by contradiction"). The tactic `SPOSE_NOT_THEN STRIP_ASSUME_TAC` moves the negation of the goal statement into the assumptions, and makes the new goal to prove false.

```
F
------------------------------------
0. oc s t v
1. ∀w. t ≠ Var w
2. wfs s
3. wfs s₂
4. v ∈ vars (s ◁ t)
5. ∀w. s ◁ t ≠ Var w
6. measure (pair_count ∘ walk* s₂) (Var v) (s ◁ t)
7. measure (pair_count ∘ walk* s) (Var v) (s ◁ t)
8. ∀v. v ∈ vars t ⇒ measure (pair_count ∘ walk* s₂) (Var v) t
9. ∀v. v ∈ vars t ⇒ measure (pair_count ∘ walk* s) (Var v) t
10. s₂ ◁ Var v = s₂ ◁ s ◁ t
```

We are now at the point where rewriting can take over. Assumption 6 says $s_2 \lhd$ `Var` v is smaller than $s_2 \lhd s \lhd t$, but assumption 10 says they are equal. We use `FULL_SIMP_TAC` to rewrite the assumptions, and give it `measure_thm` so the `measure` constant is rewritten by its definition. Automatic rewrites take care of the contradiction between the less-than implied by `measure` and the equality in assumption 10. We can now see the whole tactic that solves the initial goal.

```
STRIP_TAC THEN
`v IN vars (walk* s t)`
  by METIS_TAC [oc_eq_vars_walkstar,IN_DEF] THEN
`!w. (walk* s t) <> Var w`
  by (Cases_on `t` THEN FULL_SIMP_TAC (srw_ss()) []) THEN
IMP_RES_TAC vars_measure THEN
SPOSE_NOT_THEN STRIP_ASSUME_TAC THEN
FULL_SIMP_TAC (srw_ss()) [measure_thm]
```

The proof we have constructed for Lemma 25 is actually quite short by HOL4 standards. It does, however, show the level of detail that might need to go into a tactic.

We see a range of levels of guidance, from declaring and solving specific subgoals like $v \in$ `vars` $(s \triangleleft t)$ to simply applying `IMP_RES_TAC` and hoping a useful assumption comes out of it.

Previously proved theorems, like `oc_eq_vars_walkstar`, can make proving later theorems much easier. Coming up with useful lemmas, or indeed definitions, affects the development of a theory, just as in an unmechanised setting. Statements of lemmas most useful for subsequent proofs are often, but not always, statements that are easy to read. Not all of the theorems in the miniKanren theories will appear in this thesis, because many of them embody boring or tedious "glue" that is better summarised by the high-level proof text of a more important lemma. The lemmas that do appear have been chosen as guides towards the main results.

We won't see any more tactics in this thesis. More than once has a tactic been shortened or otherwise improved after trying to describe the proof in English, but many tactics have been left unimproved. It is the theorems themselves, and the English proofs where they are written, that are important to understanding the theories. While the tactics could be read by HOL4 experts, their main point is just to solve the goal. In any case, tactics are better understood interactively than by only reading the source code.

Mechanised theorem proving does not have to mean writing relatively unreadable tactics. So called declarative proof languages such as Isar [Wenzel 1999] have been developed for systems other than HOL4. The trade-offs between HOL style tactics and more declarative proof languages are explored in [Harrison 1996].

Chapter 2

Triangular Substitutions

2.1 Introduction

We begin with definitions of terms, substitutions, composition of substitutions, and triangular form. We will see that triangular substitutions are those than can be updated non-destructively, meaning the previous version remains accessible. This makes triangular substitutions suitable for persistent data structures, and is the main reason to use them in a logic programming system.

Substitutions can be applied to terms, and our notation for this is s " t. For triangular substitutions, however, application needs to be *repeated*, because the full effect of a triangular substitution may not be imparted by a single application. We write repeated application with a triangle: $s \triangleleft t$. The second and third sections of this chapter are about defining repeated application, which is subtle because it is not always possible. We isolate an important well-formedness condition on substitutions that enables repeated application in Section 2.2, and define repeated application in Section 2.3.

The usual focus when developing substitutions for unification are *idempotent* substitutions. We will look at the relationship between idempotent and triangular substitutions throughout the chapter. The former are a special case of the latter: idempotent substitutions are where single and repeated application coincide.

In the last section of the chapter, we consider a concrete representation of substitutions: association lists.

2.1.1 Terms

miniKanren is a first-order logic programming language, so miniKanren programs correspond to (a restricted class of) formulas in first-order logic. Formulas are made out of *terms*. From a programming language perspective, this means all the values (*i.e.*, data structures) in miniKanren are terms.

Terms are trees with variables and constants at the leaves. We restrict ourselves to binary trees and define first-order terms inductively as follows. Our terms are parameterised by the type α of the constants, and we use natural numbers for variables[1].

[1] We will more often write `Var` y than `Var 2`. The y in the first term is a HOL variable standing for a number.

Definition 1. Terms

α `term = Var of num | Pair of` α `term =>` α `term | Const of` α

Our presentation of terms is different from the usual definition, although ours has been used for mechanisation before [Slind 1999; Paulson 1985]. Traditionally, first-order terms are defined as an algebraic structure over a signature $\Sigma = (V, F, a : F \to \mathbb{N})$ where V, the variable symbols, and F, the function symbols, are disjoint countable sets, and $a(f)$ is called the arity of f. The terms are defined inductively as follows: any variable is a term; and if $f \in F$ with $a(f) = n$, and if $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term. Definition 1 terms are isomorphic to standard terms.[2] We use our definition because it more closely matches terms in miniKanren, which are represented by Scheme lists (which are made out of pairs).

The set of variables in a term is defined recursively, to match the inductive data type.

Definition 2. Variables in a term

$$\begin{aligned}
\texttt{vars}\ (\texttt{Var}\ x) &= \{x\} \\
\texttt{vars}\ \langle t_1, t_2\rangle &= \texttt{vars}\ t_1 \cup \texttt{vars}\ t_2 \\
\texttt{vars}\ (\texttt{Const}\ c) &= \{\,\}
\end{aligned}$$

We define the bag of variables in a term t, written `varb` t, similarly.

The function `pair_count` t returns the number of pairs in t. We use the number of pairs to quantify a term's size.

At times, we will use the more compact notation for terms from Chapter 1, writing $\langle\texttt{Var}\ x, \texttt{1}\rangle$, or $\langle x, 1\rangle$ when the subterm is obviously a variable, instead of `Pair (Var` x`) (Const 1)`.

2.1.2 Substitution

Substitution is the act of consistently replacing some variables in a term by new terms. The information about which variable is bound to which new term is also called a substitution. This information is usually encoded as a function from variables to terms that is almost everywhere the identity. If we leave the identity bindings implicit, then a substitution is a finite map from variables to terms. The HOL type α `subst` of substitutions in our theories is an abbreviation for a finite map from numbers to α `terms`.

[2] To recover standard terms, assuming A is the set of all elements of type α, take $V = \mathbb{N}$, $F = \{\mathrm{P}\} \uplus A$ where P is a new function symbol , $a(\mathrm{P}) = 2$, and $a(x \in A) = 0$. To construct our terms from standard terms, use F as the constant values, and assuming $V = \{v_0, v_1, \ldots\}$, take v_i to `Var` i, and take $f(t_1, \ldots, t_n)$ to `Pair` f (`Pair` $[t_1]$ $(\ldots($`Pair` $[t_{n-1}]$ $[t_n]))$.

If we lift finite map application from variables to terms, we get substitution application.

Definition 3. Substitution application

$$
\begin{aligned}
s \;\texttt{"}\; \texttt{Var}\; v &= \texttt{case}\; s\;\texttt{'}\; v\; \texttt{of NONE} \to \texttt{Var}\; v \parallel \texttt{SOME}\; t \to t \\
s \;\texttt{"}\; \langle t_1, t_2 \rangle &= \langle s \;\texttt{"}\; t_1, s \;\texttt{"}\; t_2 \rangle \\
s \;\texttt{"}\; \texttt{Const}\; c &= \texttt{Const}\; c
\end{aligned}
$$

For example, if $s = \{x \mapsto \langle y, z\rangle, z \mapsto c\}$, then $s \;\texttt{"}\; \langle x, z\rangle$ is $\langle\langle y, z\rangle, c\rangle$.

Substitutions are almost everywhere the identity, so applying a substitution to a variable outside its domain simply returns the variable. The finite map representation also permits a substitution that explicitly binds a variable to itself, for example $\{x \mapsto \langle y, z\rangle, y \mapsto c, z \mapsto z\}$. We often want to exclude such bindings, and do so with the condition `noids` s.

Definition 4. Substitutions without explicit identity bindings

$$\texttt{noids}\; s \iff \forall v.\; s\;\texttt{'}\; v \neq \texttt{SOME (Var}\; v\texttt{)}$$

A substitution is *idempotent* if repeated application is the same as a single application.

Definition 5. Idempotent substitutions

$$\texttt{idempotent}\; s \iff \forall t.\; s \;\texttt{"}\; s \;\texttt{"}\; t = s \;\texttt{"}\; t$$

Idempotent substitutions can be characterised as follows.

Lemma 1. *The domain and range of an idempotent substitution are disjoint*

$$\vdash \texttt{idempotent}\; s \wedge \texttt{noids}\; s \iff \texttt{DISJOINT (FDOM}\; s\texttt{) (rangevars}\; s\texttt{)}$$

Proof. From left to right: Suppose x is a variable in the range of the substitution. Then there exists a variable v such that $x \in \texttt{vars (}s \;\texttt{"}\; v\texttt{)}$. If x were also in the domain of the substitution, bound to t, then applying s again to $s \;\texttt{"}\; v$ would send x to t. The `noids` assumption means $t \neq \texttt{Var}\; x$, so we would contradict the idempotence assumption.

From right to left: A binding from a variable to itself would put that variable in the domain and the range, which is impossible by assumption, so the `noids` condition must be satisfied. For the idempotence conjunct, if $s \;\texttt{"}\; t$ differs from $s \;\texttt{"}\; s \;\texttt{"}\; t$, then we can show by term induction that $s \;\texttt{"}\; t$, which is clearly in the range, contains a variable in the domain of s. □

Substitutions can be viewed at different levels of abstraction, and we will distinguish at least three levels.

1. The most abstract view is concerned with the full effect of a substitution on a term. Repeated application of some substitutions reaches a fixpoint, and it is this fixpoint, if it exists, we use to distinguish substitutions at this level.

   For example, let $s_1 = \{x \mapsto \langle y,\, z\rangle, y \mapsto c_1, z \mapsto c_2\}$ and $s_2 = \{x \mapsto \langle c_1,\, c_2\rangle, y \mapsto c_1, z \mapsto c_2\}$. Then s_1 " s_1 " `Var` $x = \langle c_1,\, c_2\rangle = s_2$ " `Var` x. In fact s_2 is an idempotent version of s_1, and they are equal after full application on every term. We write $s \triangleleft t$ for the full application of s to t, and will define this function in Section 2.3.

2. The middle view is the finite map view of substitutions we started with. From this view we can distinguish s_1 and s_2 above since s_1 " `Var` $x = \langle y,\, z\rangle \neq \langle c_1,\, c_2\rangle = s_2$ " `Var` x.

   Technically there are two levels here: finite maps from variables to terms, and the substitution application function from terms to terms. But since substitution application is the homomorphic extension of finite map application, they correspond perfectly.

3. The most concrete view sees substitutions as data structures that represent finite maps.

   Suppose our data structure is the association list (*i.e.*, a list of bindings represented by pairs). Then we can distinguish $s_{11} = [(x, \langle y,\, z\rangle); (y, c_1); (z, c_2)]$ from $s_{12} = [(y, c_1); (z, c_2); (x, \langle y,\, z\rangle)]$, although they both represent the same finite map. We will examine this view in more detail in Section 2.5.

2.1.3 Composition and triangular form

The composition of substitution s_2 over s_1, denoted $s_2 \odot s_1$, is usually defined in such a way that $s_2 \odot s_1$ " t equals s_2 " s_1 " t. Computing such a composition isn't difficult, but involves applying s_2 to every term in the range of s_1. We define a function that computes the composition of a substitution with itself.

Definition 6. Application of a substitution to itself

`selfapp` s = (") $s \circ s$

This somewhat cryptic definition says `selfapp` s is the finite map obtained by composing a function, one step application of s, over the finite map s. So every term in the range of s is replaced by its image under s. A straightforward induction on terms shows that this is a composition.

Lemma 2. $\vdash$ `selfapp` s " $t = s$ " s " t

Self application is the furthest we will go towards computing compositions in this manner. When we only need the effect of composition, we can compose application functions directly: s_2 " s_1 " t. When we need the composition, $s_2 \odot s_1$, using *triangular substitutions* gives us an alternative.

In their chapter on Unification Theory [Baader and Snyder 2001], Baader and Snyder describe the triangular form of a substitution as a sequential list of its bindings. This means every substitution has a triangular form, which is simply its representation as a finite map: our middle level of abstraction.[3] The term "triangular substitution" has more meaning, however, as an indication of how substitutions are being used. Specifically, two substitutions are triangular if we can compose them without changing any of the bindings in either substitution.

To compose triangular substitutions, we simply take the union of their bindings. For example, the composition of $s_1 = \{x \mapsto \texttt{Var}\ \ y, y \mapsto c\}$ and $s_2 = \{w \mapsto \texttt{Var}\ \ y\}$ is $s_3 = \{w \mapsto \texttt{Var}\ \ y, x \mapsto \texttt{Var}\ \ y, y \mapsto c\}$. This operation is not composition in the sense of one step application: s_3 " $\texttt{Var}\ \ w = \texttt{Var}\ \ y$ whereas s_1 " s_2 " $\texttt{Var}\ \ w = c$. For triangular substitutions, we need to consider full application instead, *i.e.*, $s_3 \lhd \texttt{Var}\ \ w$, which does equal $s_1 \lhd s_2 \lhd \texttt{Var}\ \ w$.

When non-triangular substitutions are composed, by contrast, the result may contain bindings that don't appear in either of the original substitutions. The non-triangular composition of s_1 and s_2 above is $\{w \mapsto c, x \mapsto \texttt{Var}\ \ y, y \mapsto c\}$, and $w \mapsto c$ is new. The idempotent composition, which additionally ensures that the result is idempotent, is $\{w \mapsto c, x \mapsto c, y \mapsto c\}$.

To compose triangular substitutions that bind the same variable we can, following [Baader and Snyder 2001], drop any binding in the second substitution of a variable already bound by the first before taking the union. For example $\{x \mapsto c_2, y \mapsto c_1\} \odot \{w \mapsto \texttt{Var}\ \ x, x \mapsto c_1\}$ becomes $\{y \mapsto c_1\} \odot \{w \mapsto \texttt{Var}\ \ x, x \mapsto c_1\}$ and the result is $\{w \mapsto \texttt{Var}\ \ x, x \mapsto c_1, y \mapsto c_1\}$. Alternatively, we can simply forbid composition of triangular substitutions that bind the same variable, allowing only non-triangular composition in that case. This is reasonable, since in our unification algorithms we will only need to compose triangular substitutions with disjoint domains (in fact, one of them will always be a singleton map).

Etymology Describing substitutions that are composed by taking the union of their bindings as triangular is a curious custom. The origin of the term is the triangular form of a system of linear equations, which can be reached by Gaussian elimination (when the system has a unique solution). If we impose an ordering on the variables $x_1, \ldots, x_n$ in a system of linear equations, then we can say the system is in triangular form if the ith equation refers to the variables $x_i, \ldots, x_n$ only. Thus the following system is in triangular form.

$$\begin{aligned} 2x + y - z &= 8 \\ 2y + z &= 1 \\ -z &= 2 \end{aligned}$$

[3] Actually a list would be at our concrete level, but since Baader and Snyder appear to ignore the order of the list and exclude multiple bindings for the same variable, they are really maintaining a finite map.

The triangle is now visible. The relationship between the equations above and a substitution is that each equation can be seen as a binding for its first variable. Thus $x \mapsto \frac{1}{2}(8 - y + z)$, $y \mapsto \frac{1}{2}(1 - z)$, and $z \mapsto -2$. The crucial condition that allows us to write any system of equations in this way, with fewer and fewer variables mentioned from top to bottom, is that there are no cyclic dependencies in the bindings. This is exactly the well-formedness condition we impose on triangular substitutions in Section 2.2.

2.1.4 Persistence and search trees

Composing triangular substitutions leaves the original substitutions intact. This is the advantage of using triangular substitutions. A data structure is *persistent* if previous versions of the data structure are still accessible after updates. Triangular substitutions are apt to be implemented as persistent data structures.

Persistence is of particular interest in systems that do a branching search, because earlier versions of data structures are updated in alternative ways that all need access to the information in the earlier version. The use of persistent data structures allows that information to be shared, and this can mean good time and memory efficiency for the overall search.

```
(run* (q)
  (exist (w x y z)
    (≡ q ⟨w, ⟨x, z⟩⟩)
    (≡ q ⟨x, y⟩)
    (conde ((≡ x 1)) ((≡ x 2)))
    (conde ((≡ y ⟨1, x⟩)) ((≡ y ⟨x, 2⟩)))))
⇒ (⟨1, ⟨1, 1⟩⟩ ⟨2, ⟨2, 2⟩⟩ ⟨1, ⟨1, 2⟩⟩)
```

Listing 2.1: A program to illustrate branching search.

We will illustrate the idea of branching search in LP with an extended example. Consider the miniKanren program in Listing 2.1. The corresponding formula is

$$\begin{aligned} \exists q\, w\, x\, y\, z.\, q = \langle w, \langle x, z\rangle\rangle \wedge \\ q = \langle x, y\rangle \wedge \\ (x = 1 \vee x = 2) \wedge \\ (y = \langle 1, x\rangle \vee y = \langle x, 2\rangle) \end{aligned}$$

Figure 2.1 on page 25 is an illustration of the search miniKanren would use to solve this program. The top part of each box is a unification problem appearing in Listing 2.1. In a recursive program, the tree might have more than one box for different instances of the same unification problem, but in this case there is no recursion. The bottom part lists the bindings added to the substitution if unification succeeds. At any box in the tree, the *current substitution* is the union of the bindings in that box and all its

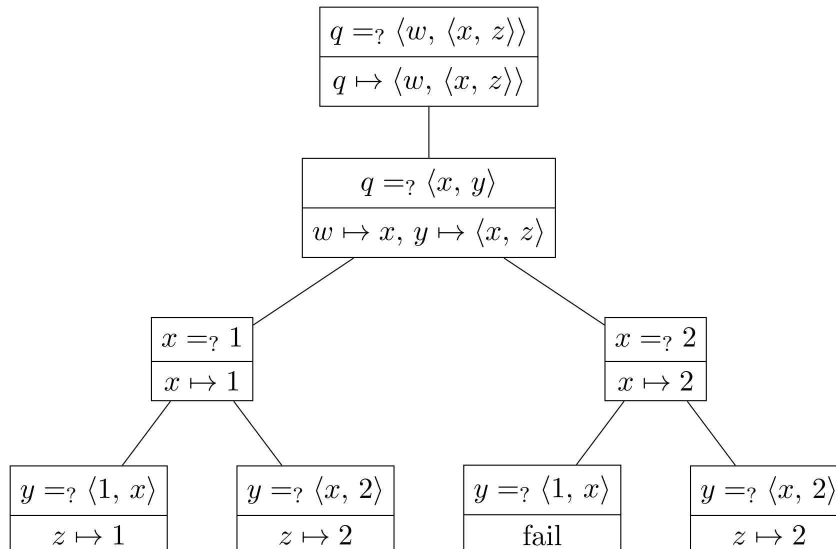


Figure 2.1: Search tree for a miniKanren program. Each box contains a unification problem from Listing 2.1 and the bindings forming its solution, if there is one. The union of bindings in any path gives the current substitution at the path's end. Older (higher) versions of the current substitution are revisited during a tree traversal, and they don't need to be destroyed to make the newer versions.

ancestors. Thus bindings near the top of the tree are shared by the substitutions at the bottom.

Branches in the tree correspond to disjunction. Older versions of the current substitution are extended in a different way as the search moves from one branch to another. Thus it makes sense to maintain a partial (*i.e.*, older) substitution, such as

$$\{q \mapsto \langle w,\ \langle x,\ z\rangle\rangle, w \mapsto x, y \mapsto \langle x,\ z\rangle\}$$

and share it between the different extensions. The extended substitution for the leftmost child of the tree is

$$\{q \mapsto \langle w,\ \langle x,\ z\rangle\rangle, w \mapsto x, y \mapsto \langle x,\ z\rangle, x \mapsto 1, z \mapsto 1\}$$

The alternative would be to store a complete, idempotent substitution at each node, such as

$$\{q \mapsto \langle 1,\ \langle 1,\ 1\rangle\rangle, w \mapsto 1, y \mapsto \langle 1,\ 1\rangle, x \mapsto 1, z \mapsto 1\}$$

for the leftmost child. Ultimately, we want to know the full, idempotent effect of the current substitution on the terms we're trying to unify. Using triangular, shared, substitutions moves the work of calculating this full effect. In the triangular setting, the work is done during substitution application (using the full version $s \triangleleft t$ rather than s " t). In the idempotent setting, the work is done when ensuring the result of a successful unification is idempotent.

Although substitution application may be more frequent than unification, the ben-

efit of using persistent data structures may outweigh the cost of a more involved application function. We have some evidence of this from experiments comparing the efficiency of idempotent and triangular substitutions in αKanren, which are described in Section 4.5. It is also telling that in a comparison of unification algorithms for use in automated theorem proving, Hoder and Voronkov [Hoder and Voronkov 2009] used triangular substitutions exclusively. The different unification algorithms, which we describe in Section 6.2, are often presented so as to produce idempotent results, but perhaps in practice they are modified for triangular usage.

2.2 Well-formedness

The purpose of the well-formedness condition is to ensure that full application $(s \lhd t)$ is possible. Sometimes it is not possible: if we take $s = \{x \mapsto y, y \mapsto \langle c, \texttt{Var}\ x\rangle\}$, then it is not clear what $s \lhd \texttt{Var}\ x$ should be, since our terms are finite. Intuitively, the problem is that s has a cyclic chain of bindings. We would like such a substitution to be malformed.

We can capture well-formedness formally as follows. For each substitution s we define a relation $\texttt{tri}_\texttt{R}\ s$ that holds between a variable in the domain and a variable in the corresponding term.

Definition 7. Relating a variable to those in the term to which it's bound

$$\texttt{tri}_\texttt{R}\ s\ y\ x \iff \texttt{case}\ s\ \texttt{'}\ x\ \texttt{of NONE} \to \texttt{F} \parallel \texttt{SOME}\ t \to y \in \texttt{vars}\ t$$

For example, if $s = \{x \mapsto \langle \texttt{Var}\ y, \texttt{Var}\ z\rangle, w \mapsto \texttt{Var}\ x\}$ then the following relationships hold: $\texttt{tri}_\texttt{R}\ s\ y\ x$, $\texttt{tri}_\texttt{R}\ s\ z\ x$, and $\texttt{tri}_\texttt{R}\ s\ x\ w$ (and hence $(\texttt{tri}_\texttt{R}\ s)^+\ y\ w$, *etc.*). A substitution is *well-formed*, written $\texttt{wfs}\ s$, if $\texttt{tri}_\texttt{R}\ s$ is well-founded.

We will characterise well-formedness in three ways. The first says that well-formedness is equivalent to having no cycles from a variable to itself.

Lemma 3. *Only well-formed substitutions have no cycles*

$$\vdash \texttt{wfs}\ s \iff \forall v.\ \neg(\texttt{tri}_\texttt{R}\ s)^+\ v\ v$$

Proof. From left to right is easy: the transitive closure of a well-founded relation is well-founded, and well-founded relations are not reflexive.

From right to left we prove by contradiction. If s is not well-formed, there is an infinite chain of variables linked by $\texttt{tri}_\texttt{R}\ s$. By definition of $\texttt{tri}_\texttt{R}$, every variable in this chain must be in the domain of s. But the domain of s is finite, so the chain must contain a repeat, contradicting the assumption $\forall v.\ \neg(\texttt{tri}_\texttt{R}\ s)^+\ v\ v$. $\square$

Corollary 1. $\vdash \texttt{wfs}\ s \Rightarrow \texttt{noids}\ s$

Corollary 2. *Idempotent substitutions are well-formed*

$\vdash$ `idempotent` s $\wedge$ `noids` s $\Rightarrow$ `wfs` s

Proof. From Lemmas 1 and 3, since $(\texttt{tri}_{\texttt{R}}\ s)^+$ requires the variable at one end to be in the domain of the substitution and the variable at the other end to be in the range. □

The next well-formedness characterisation, Lemma 5 below, requires a preliminary result to describe how $\texttt{tri}_{\texttt{R}}\ (\texttt{selfapp}\ s)$ relates to $\texttt{tri}_{\texttt{R}}\ s$. We begin by defining a sub-relation of $\texttt{tri}_{\texttt{R}}\ s$ that only allows one step.

Definition 8. "Last link only" version of $\texttt{tri}_{\texttt{R}}$

$$\texttt{tri}^1_{\texttt{R}}\ s\ y\ x \iff \texttt{tri}_{\texttt{R}}\ s\ y\ x \wedge y \notin \texttt{FDOM}\ s$$

The relation $\texttt{tri}_{\texttt{R}}\ (\texttt{selfapp}\ s)$ on the self-applied substitution consumes an even number of steps of the relation $\texttt{tri}_{\texttt{R}}\ s$ on the original substitution, because self-application essentially halves all variable chains through the substitution. An odd number of steps indicates that the chain has stopped on a variable no longer in the domain.

Lemma 4. $(\texttt{tri}_{\texttt{R}}\ (\texttt{selfapp}\ s))^+$ *either consumes an even number of* $\texttt{tri}_{\texttt{R}}\ s$ *steps, or stops on a variable outside the domain*

$$\begin{aligned}&\vdash (\texttt{tri}_{\texttt{R}}\ (\texttt{selfapp}\ s))^+\ v\ u \iff \\ &\quad (\exists n.\ \texttt{NRC}\ (\texttt{tri}_{\texttt{R}}\ s)\ (2 * \texttt{SUC}\ n)\ v\ u) \vee \\ &\quad \exists n\ u'.\ \texttt{NRC}\ (\texttt{tri}_{\texttt{R}}\ s)\ (2 * n)\ u'\ u \wedge \texttt{tri}^1_{\texttt{R}}\ s\ v\ u'\end{aligned}$$

Proof. From right to left follows by induction on n. From left to right follows by induction on the number of steps in the transitive closure. □

Lemma 5. *Only well-formed substitutions are well-formed after self-application*

$\vdash$ `wfs` $s \iff$ `wfs (selfapp` s`)`

Proof. From Lemmas 3 and 4. Any cycle in the original substitution can be made into a cycle of even length (by repeating if necessary), and therefore also appears in the self-applied substitution. Conversely, any cycle in the self-applied substitution is clearly made out of a cycle in the original substitution twice as long. (It can't have an odd length since that would require the variable at both ends of the cycle to be both in the domain and not in the domain.) □

Finally, our third characterisation of well-formed substitutions.

Theorem 1. *Only well-formed substitutions have fixpoints*

$$\vdash \texttt{wfs}\ s \iff \exists\, n.\ \texttt{idempotent (selfapp}^n\ s\texttt{)} \land \texttt{noids (selfapp}^n\ s\texttt{)}$$

Proof. From right to left, the result follows by induction on n. From left to right, the `noids` condition follows from Lemma 5 and Corollary 1. Idempotence follows by contradiction. If a substitution is not idempotent there will be a variable that maps to a term including a variable in the substitution's domain. If this occurs within a substitution iterated n times, there must be a chain of length n within the original substitution with the same property. But an arbitrarily long chain cannot exist without a loop, contradicting our well-formedness assumption. □

Theorem 1 (with Lemma 2) shows that well-formedness is necessary and sufficient for being able to recover an equivalent idempotent substitution by repeated application. We will see this again more concretely in Section 2.4.

2.3 Substitution application

In this section we formally define the function that computes the full application of a substitution to a term. Not all triangular substitutions are idempotent, but full application allows us to treat a substitution as if we had collapsed it down to an idempotent one by repeated self-application, without actually doing so. This is only possible when the substitution is well-formed.

Our application function is called `walk*`; the notation $s \triangleleft t$ means application of `walk*` to substitution s and term t. The collapsing effect of `walk*` is achieved by recursion: if we encounter a variable in the domain of the substitution, we look it up and recur on the result. Defining this function presents the first of a number of interesting termination problems, which we will see in Section 2.3.2.

The clearest expression of `walk*`'s behaviour is the following characterisation:

Lemma 6. *Characterisation of* `walk*`

$$\begin{array}{l} \vdash \texttt{wfs}\ s \Rightarrow \\ \quad s \triangleleft \texttt{Var}\ v = \texttt{(case}\ s\ \texttt{'}\ v\ \texttt{of NONE} \rightarrow \texttt{Var}\ v \parallel \texttt{SOME}\ t \rightarrow s \triangleleft t\texttt{)} \land \\ \quad s \triangleleft \langle t_1, t_2\rangle = \langle s \triangleleft t_1, s \triangleleft t_2\rangle \land s \triangleleft \texttt{Const}\ c = \texttt{Const}\ c \end{array}$$

The clauses we use to define `walk*` will be in terms of a helper function `walk`.

2.3.1 `walk`

The `walk*` function can be viewed as performing a tree traversal of its eventual output term. Other algorithms, including `unify`, need to perform some of this tree walk, but may not need to immediately traverse a term to its leaves. We isolate the part of `walk*` that finds the ultimate binding of a variable, calling this `vwalk`:

Definition 9. Walking a variable

```
wfs s ⇒
  vwalk s v =
    case s ' v of
      SOME (Var u) → vwalk s u
    || SOME t → t
    || NONE → Var v
```

Proving termination for `vwalk` under the assumption `wfs` s is straightforward. The substitution doesn't change, so can be treated schematically (*i.e.*, ignored for the termination argument). We use the termination relation $\mathtt{tri_R}\ s$, which clearly holds for the recursive call in `vwalk`, and we simply assume it is well-founded.

To better match the miniKanren code, we define a function `walk` on terms, which either calls `vwalk` if its argument is a variable, or returns its argument. It is a common miniKanren idiom (used in `unify`, among other places) to begin functions by walking term arguments in the current substitution. This reveals just enough of a term-in-context's structure for the current level of recursion. This idiom is used in the definition of `walk*`, which can be stated thus:

Definition 10. Substitution application, walking version

```
wfs s ⇒
  s ◁ t = case walk s t of ⟨t₁, t₂⟩ → ⟨s ◁ t₁, s ◁ t₂⟩ || t′ → t′
```

Lemma 6 follows from this definition by recursion induction on `vwalk`.

2.3.2 Termination of `walk*`

HOL4 generates the following termination conditions when we attempt to make Definition 10 (without the `wfs` s hypothesis), representing the need for a well-founded relation that holds on every recursive call. Our goal is then to find a relation R that reduces these conditions to well-formedness of the input substitution.

1. $\mathtt{WF}\ R$
2. $\forall t\ t_1\ t_2.\ \mathtt{walk}\ s\ t = \langle t_1, t_2\rangle \Rightarrow R\ t_1\ t$
3. $\forall t\ t_1\ t_2.\ \mathtt{walk}\ s\ t = \langle t_1, t_2\rangle \Rightarrow R\ t_2\ t$

The termination relation we use is the lexicographic combination of the multi-set ordering with respect to $(\mathtt{tri_R}\ s)^+$ over a term's variables, and the term's size.

Definition 11. Termination relation for $\texttt{walk}^*$

$$\texttt{walk}^*{}_{\texttt{TR}}\ s\ =\ \texttt{inv_image (mlt (tri}_{\texttt{R}}\ s\texttt{)}^{+}\ \texttt{LEX measure pair_count) (}\lambda\, t\texttt{. (varb}\ t\texttt{,}t\texttt{))}$$

Lemma 7. $\texttt{walk}^*{}_{\texttt{TR}}$ *is well-founded*

$$\vdash \texttt{wfs}\ s \Rightarrow \texttt{WF (walk}^*{}_{\texttt{TR}}\ s\texttt{)}$$

Proof. Since any measure relation is well-founded, and since inverse image, lexicographic combination, multi-set ordering, and transitive closure all preserve well-foundedness. □

Lemma 8. *Termination Condition 2*

$$\vdash \texttt{wfs}\ s \Rightarrow \forall\, t\ t_1\ t_2.\ \texttt{walk}\ s\ t = \langle t_1, t_2\rangle \Rightarrow \texttt{walk}^*{}_{\texttt{TR}}\ s\ t_1\ t$$

Proof. Since t walks to a pair, t is either the pair itself or a variable.

If t is the pair, then t_1 has no more variables than t (the difference is the variables in t_2), and t_1 is smaller than t. Therefore either the $\texttt{mlt (tri}_{\texttt{R}}\ s\texttt{)}^{+}$ half of the lexicographic combination will succeed by replacing any extra variables by the empty bag, or if there are no extra variables, the `measure pair_count` half will succeed.

If t is a variable, the $\texttt{mlt (tri}_{\texttt{R}}\ s\texttt{)}^{+}$ half of the lexicographic combination is satisfied, since $\texttt{(tri}_{\texttt{R}}\ s\texttt{)}^{+}$ relates the variables in the input of a walk to those in the output. Thus the variable t can be replaced by the bag of variables in $\langle t_1, t_2\rangle$ and for each variable v in the replacement we will have $\texttt{(tri}_{\texttt{R}}\ s\texttt{)}^{+}\ v\ t$. □

The proof of Condition 3 is similar.

2.4 Collapsing triangular substitutions

In Section 2.2 we saw that well-formed substitutions can be collapsed into equivalent idempotent substitutions. We subsequently defined $\texttt{walk}^*$ to perform this collapse. In this section, we develop some properties of $\texttt{walk}^*$, and show that it meets its design goal.

Our first lemma says that applying a submap of a substitution, and then the larger substitution, is the same as applying the larger substitution alone. This lemma justifies our calling application of $\texttt{walk}^*$ the "full application". All the relevant information in the substitution is imparted to the term at once.

Lemma 9. $\texttt{walk}^*$ *over a submap*

$$\vdash s \sqsubseteq sx \wedge \texttt{wfs}\ sx \Rightarrow sx \triangleleft t = sx \triangleleft s \triangleleft t$$

Proof. By recursion induction on $\texttt{walk}^*$. □

Corollary 3. $\texttt{walk}^*$ *with a fixed substitution is idempotent*

We should be able to construct an idempotent equivalent to any well-formed substitution s by applying `walk*` s to every term in the range. We call the resulting substitution the *collapse* of s.

Definition 12. Collapsing a triangular substitution to idempotence

`collapse` s = `FUN_FMAP` ($\lambda\, v.\ s \lhd$ `Var` v) (`FDOM` s)

The following lemma then follows easily by recursion induction on `walk*`.

Lemma 10. `walk*` *equals application of the collapsed substitution*

$\vdash$ `wfs` $s \Rightarrow \forall\, t.$ `collapse` s `"` $t = s \lhd t$

Lemma 11. *Collapsed substitutions are idempotent*

$\vdash$ `wfs` $s \Rightarrow$ `idempotent` (`collapse` s)

Proof. By Corollary 3 and Lemma 10. □

Lemma 12. *Idempotent substitutions are collapsed*

$\vdash$ `idempotent` $s \wedge$ `noids` $s \Rightarrow$ `collapse` $s = s$

Proof. From Corollary 2 and Lemma 1 we know s is well-formed and its domain and range are disjoint. We can then show by recursion induction on `walk*` that applying `walk*` s to any term in the range leaves the term unchanged. This proves that the collapsed substitution has the same range as the original (and its domain is the same by definition). □

Lemma 13. `walk*` *reduces to application on idempotent substitutions*

$\vdash$ `wfs` $s \Rightarrow$ (`idempotent` $s \iff$ `walk*` s = (`"`) s)

Proof. By Lemma 10 we have s = `collapse` s. From left to right is by Lemmas 10 and 12 (and Corollary 1).

From right to left Lemma 11 says it's sufficient to show s = `collapse` s, and this follows from Lemma 10. □

2.5 Association lists

Substitutions in miniKanren are implemented as association lists representing finite maps. In this section we look at the relationship between association lists and finite maps. HOL4's technology for emitting executable SML code allows one to emit finite maps directly (they become a list-like abstract data type in SML), so considering implementability, finite maps are not necessarily more abstract than lists. However, algorithms working on lists, and other concrete representations, have more flexibility both to be wrong (hence benefiting from verification) and to be more efficient by exploiting information in the representation (the order of the bindings, in the case of lists).

We define functions to convert an association list into a finite map, and vice versa. To get a finite map, we fold down the list from right to left, thereby ensuring the earliest binding of each variable is used (in case there are multiple bindings). Using the first binding in an association list is a convention in Scheme. To get a list from a finite map, we map a function that returns a binding pair given a variable over a list of the variables in the domain of the finite map. The order of the variables in the list is, of course, not specified by the finite map itself; the function `SET_TO_LIST` puts them in some arbitrary order.

Definition 13. `alist_to_fmap` s = `FOLDR` ($\lambda\,(k,v)\ f.\ f$ `|+` (k,v)) `FEMPTY` s

Definition 14. `fmap_to_alist` s = `MAP` ($\lambda\,k.\ (k, s$ ' $k)$) (`SET_TO_LIST` (`FDOM` s))

The lookup function for association lists is a straightforward recursion down the list.

Definition 15. Lookup in an association list

`ALOOKUP [] ` q = `NONE`
`ALOOKUP` $((x,y)::t)$ q = if $x = q$ then `SOME` y else `ALOOKUP` t q

We can prove that lookup in an association list corresponds to lookup in a finite map.

Lemma 14. `ALOOKUP` *corresponds to* (')

⊢ `ALOOKUP` al = (') (`alist_to_fmap` al) ∧
(') fm = `ALOOKUP` (`fmap_to_alist` fm)

Proof. The first conjunct is by recursion induction on `ALOOKUP`. The second conjunct is by finite map induction. □

Now we can verify that our conversion functions are faithful, *i.e.*, that `alist_to_fmap` inverts `fmap_to_alist`.

Lemma 15. *Conversion to then from an association list preserves the finite map*

⊢ fm = `alist_to_fmap` (`fmap_to_alist` fm)

Proof. We first observe that domains of both sides are equal: `fmap_to_alist` returns a list of pairs with each element from the finite map's domain in the left hand side of some pair, and we can show by induction on the list that `alist_to_fmap` puts all elements from the left hand sides of the pairs into the domain. Then equality under finite map application follows from Lemma 14. □

We can't expect an inverse in the other direction, since that would require the order of bindings in the list to be specified. In this sense, finite maps are more abstract than association lists.

We define `vwalk_al` al as an abbreviation for `vwalk` (`alist_to_fmap` al). It is a straightforward consequence of Lemmas 15 and 14 that `vwalk_al` has the same

functional characterisation as `vwalk` (*i.e.*, Definition 9), except with `(')` replaced by `ALOOKUP`. The following is another easy consequence, and establishes the second half of the equivalence between `vwalk` and `vwalk_al` (the first half is the abbreviation itself).

Lemma 16. *Equivalence of* `vwalk` *and* `vwalk_al`

$\vdash$ `vwalk` s = `vwalk_al (fmap_to_alist` s`)`

2.5.1 The right-hand-side check

We now look at an optimisation of `vwalk_al`, the right-hand-side check, that can only be defined for the concrete association list representation, since it optimises the lookup function, which is atomic for finite maps. Our formal theory of the right-hand-side check does not cover anything except definition and termination. In particular, we leave correctness to future work. The termination proof, however, is interesting.

The right-hand-side checking walk function, `vwalk_rhs` (Definition 16), doesn't call `ALOOKUP`, but incorporates the association-list lookup into its own definition. This enables the optimisation: if the variable being looked for is on the right-hand-side of the current pair in the list, then stop immediately and declare that the variable is not bound (by returning it). If the variable is found on the left-hand-side of a pair, however, then it is bound, and we restart the walk on whatever it is bound to. To be able to restart, we carry the original substitution around in the first argument of `vwalk_rhs`; the second argument should always be a list-tail of the first (assuming at the initial call both arguments are equal).

Definition 16. Walk with right-hand-side check

```
vwalk_rhs s [] v = Var v ∧
vwalk_rhs s ((x,Var u)::t) v =
  (if u = v then
     Var v
   else
     if x = v then vwalk_rhs s s u else vwalk_rhs s t v) ∧
vwalk_rhs s ((x,y)::t) v = if x = v then y else vwalk_rhs s t v
```

Although it is not primitive recursive, `vwalk_rhs` terminates on all inputs, even when the second argument is not a list tail of the first. The behavior of the algorithm can be divided into two phases. In the first phase, we walk down the list in the second argument, looking either for a binding for v, or a pair with `Var` v on the right-hand-side. If we reach the end of the list, or find a pair with `Var` v on the right, the algorithm terminates. If we find a binding for v, we enter the second phase, which consists of a series of shorter and shorter walks each starting at the beginning of the substitution.

In the second phase, we know the second substitution is a list tail of the first, because we initialised the arguments with the call `vwalk_rhs` s s u. We also know that there exists a binding with our variable (u, which becomes the new v) on the right-hand-side, because we only enter the second phase when we find such a binding.

A binding with the sought variable on the right-hand-side guarantees termination of the algorithm: if we reach it while walking down the list, we stop; if we jump back to the start of the list before reaching it, then we will have jumped from another binding, closer to the start of the list, with the new sought variable on the right-hand-side.

We define a function `ELENGTH` (Definition 17) to measure the distance to the closest binding with the sought variable on the right-hand-side (or to the end of the list if there are none). This is the effective length of the list for `vwalk_rhs`, since the algorithm won't ever look past such a binding.

Definition 17. "Effective length" function used for `vwalk_rhs`'s termination measure

```
ELENGTH (v,[]) = 0
ELENGTH (v,h::t) = if SND h = Var v then 0 else 1 + ELENGTH (v,t)
```

Definition 18. Termination relation for `vwalk_rhs`

```
inv_image
  (measure (λ b. if b then 0 else 1) LEX measure ELENGTH LEX
   measure LENGTH)
  (λ (s0,s1,v).
     ((∃ pr.
          s0 = pr ++ s1 ∧
          ∀ e. MEM e pr ⇒ FST e ≠ v ∧ SND e ≠ Var v),(v,s0),s1))
```

The termination relation is given in Definition 18. The separation into two phases is encoded by the first component of the lexicographic combination, which will be `1` in phase 1 and `0` in phase 2. The second component of the lexicographic combination measures the effective length of the original substitution (first argument of `vwalk_rhs`), which goes down every time the algorithm jumps back to the start of the list in phase 2. The final component measures the length of the second argument of `vwalk_rhs`, which goes down while walking down the list in search of the next pair containing the sought variable.

The right-hand-side checking walk is not the same function as `vwalk_al`: it returns different answers on certain inputs. For example, take the substitution s = `[(`y`,Var `x`); (`x`,Const 3)]` and consider walking x. We get `vwalk_al` s x = `Const 3` but `vwalk_rhs` s s x = `Var` x. The problem is that s has a binding to `Var` x that appears earlier than a binding from x to another term. It should be possible to prove that the unification algorithm we define in the next chapter (redefined to work on association lists) never generates substitutions like s. This would allow us to use `vwalk_rhs` in place of `vwalk_al`, reaping any gains in efficiency without compromising correctness; we leave this to future work. We would be trading the well-formedness condition currently required to use `vwalk` for a different condition on substitutions as lists.

Chapter 3

First-Order Unification

3.1 Introduction

The question answered by unification of first-order terms is “Can these terms be made equal?”. Making terms equal means finding bindings for the variables such that after substitution the terms are identical. For example, the terms $t_1 = \langle \texttt{Var}\ x, \texttt{Var}\ z\rangle$ and $t_2 = \langle c, \texttt{Var}\ y\rangle$ can be made equal by the substitution $s_1 = \{x \mapsto c, y \mapsto \texttt{Var}\ z\}$. Applying s_1 to either term yields $\langle c, \texttt{Var}\ z\rangle$. For miniKanren we restrict our attention to the binary unification problem, which asks whether two terms can be made equal; in general, a unification problem can be over any finite set of terms. Surveys of unification can be found in [Knight 1989] and [Baader and Snyder 2001].

It is not always possible to make terms equal. For example `Const 1` and `Const 2` can never be equal, so they do not unify. Similarly, unification would fail on $\langle \texttt{Var}\ x, \texttt{Const 1}\rangle$ and $\langle \texttt{Var}\ y, \texttt{Const 2}\rangle$. The first goal of a unification algorithm is to determine whether unification is possible, and if it is possible, the second goal is to return a unifying substitution.

A unifying substitution, when it exists, is not necessarily unique. The substitution $s_2 = \{x \mapsto c, z \mapsto \texttt{Var}\ y\}$ is also unifier of t_1 and t_2. We have s_2 " $t_1 = \langle c, \texttt{Var}\ y\rangle =$ s_2 " t_2. Yet another unifier is $s_3 = \{x \mapsto c, y \mapsto c, z \mapsto c\}$, which sends both terms to $\langle c, c\rangle$. However, s_3 is *less general* than either s_1 or s_2, because it specifies bindings for y and z that aren't necessary. A unifying substitution s is more general than another unifier r if r is the composition of s with some other substitution. In our example, $s_3 = \{z \mapsto c\}s_1 = \{y \mapsto c\}s_2$. The third goal of a unification algorithm is to return, on success, a most general unifier. Staying most general prevents later unification problems from being rendered unsolvable by unnecessary bindings introduced in an earlier problem.

Given our interest in triangular substitutions, we will also consider *triangular unifiers*, which are well-formed and make terms equal under full application. Substitutions s_1 through s_3 are all triangular unifiers of t_1 and t_2, but so is $s_4 = \{z \mapsto c, y \mapsto \texttt{Var}\ w, x \mapsto \texttt{Var}\ w, w \mapsto c\}$. We have $s_4 \lhd t_1 = \langle c, c\rangle = s_4 \lhd t_2$. Generality of triangular unifiers is described in Section 3.5.2. In the triangular setting, it makes sense to talk about unification in the context of some initial substitution. Unifying t_1 and t_2 in the context of a substitution $s_0 = \{x \mapsto \texttt{Var}\ w, w \mapsto c\}$ is equivalent to unifying $\langle c, \texttt{Var}\ z\rangle$ and $\langle c, \texttt{Var}\ y\rangle$ in the empty context. The substitution s_4 might have been

obtained by unifying t_1 and t_2 in an initial context of $\{y \mapsto \texttt{Var}\ \ w, x \mapsto \texttt{Var}\ \ w, w \mapsto c\}$, of which s_4 is a triangular extension.

In the next section, we discuss *sound* unification, which produces only well-formed unifiers. The middle of the chapter is divided into three parts, and is about the unification algorithm in miniKanren that works with triangular substitutions. We first look at the definition of the algorithm, then the proof of its termination, and finally statements and proofs of its correctness. In the last section, we compare our triangular substitution algorithm to some other approaches to unification that maintain idempotent substitutions.

3.2 The occurs check

Unification of finite terms, like the ones we have defined, cannot permit the binding of a variable to a term containing itself. A substitution binding x to $\langle \texttt{Var}\ \ x, c\rangle$, say, apart from not being well-formed, can reasonably be interpreted as binding x to the infinite term $\texttt{Pair (Pair (Pair}\ \ldots\ c\texttt{)}\ c\texttt{)}\ c$. But since our terms are finite, we say x cannot be equal to $\langle \texttt{Var}\ \ x, c\rangle$ and disallow the binding. The simplest way to avoid introducing such a binding during unification is to check for occurrence of the variable in the term just before making a binding; our unification algorithms do this check.

We define the occurs check first as an inductive relation (Definition 19).

Definition 19. Occurs check relation

$$\frac{v \in \texttt{vars}\ t \quad v \notin \texttt{FDOM}\ s}{\texttt{oc}\ s\ t\ v} \qquad \frac{u \in \texttt{vars}\ t \quad \texttt{vwalk}\ s\ u = t' \quad \texttt{oc}\ s\ t'\ v}{\texttt{oc}\ s\ t\ v}$$

In a non-triangular setting, the occurs check would simply be $v \in \texttt{vars}\ \ t$, but the assumption would be that the current substitution s had already been fully applied to the term. Our occurs check relation is in the context of an input substitution s. The first rule of the relation is a base case, and says v occurs in t with respect to s if v is literally in t and is not in the domain of s. The other rule is an inductive case. If v occurs in a term t' with respect to s, and there is a variable u in t that walks to t', then v also occurs in t with respect to s.

The "walk first" idiom used for `walk*` is used again in the functional characterisation of `oc`, which matches the definition in the Scheme implementation of miniKanren.

Lemma 17. *Occurs check, walking version*

```
⊢ wfs s ⇒
    (oc s t v ⟺
       case walk s t of
         Var u → v = u
       ‖ ⟨t1, t2⟩ → oc s t1 v ∨ oc s t2 v
       ‖ _ → F)
```

Proof. The pair and constant cases (of t) are straightforward. For the `Var` case, we prove from left to right by recursion induction on `vwalk`, and from right to left by rule induction on `oc`. □

The following lemma makes the meaning of the input substitution clearer: it is the substitution we are considering all terms to be fully applied under.

Lemma 18. *The occurs check finds variables in the term after application*

```
⊢ wfs s ⇒ (oc s t v ⟺ v ∈ vars (s ◁ t))
```

Proof. From left to right, by recursion induction on `walk*`. From right to left, by rule induction on `oc`. □

3.3 Definition

Our unification algorithm, `unify`, has type

```
α subst → α term → α term → α subst option
```

The option type in the result is used to signal whether or not the input terms are unifiable. We accept that `unify` will have an undefined value when given a malformed substitution as input. Our strategy for defining `unify` is to define a total version, `tunify`; to extract and prove the termination conditions; and to then show that `unify` exists and equals `tunify` for well-formed substitutions. The definition of `tunify` is given in Figure 3.1 on page 38.

We use the "walk-first" idiom again in `unify`, walking both input terms in the input substitution. On success, the result of `unify` will be an extension of the input substitution (we will prove this in Section 3.5.1). Algorithms which take a partial version of their eventual output as input are said to be in accumulator-passing style; in our case, `unify` accumulates bindings in the substitution argument.

The case analysis in the definition of `tunify` can be understood as follows. If unification is to succeed, the (walked) terms must be the same shape (*i.e.*, produced by the same constructor), or one of the terms must be a variable. The first three patterns in the case expression are for when one of the terms is a variable: if both terms are the

Definition 20. Unification with triangular substitutions (total version)

```
tunify s t1 t2 =
  if wfs s then
    case (walk s t1,walk s t2) of
      (Var v1,Var v2) →
        SOME (if v1 = v2 then s else s |+ (v1,Var v2))
    ‖ (Var v1,t2) → if oc s t2 v1 then NONE else SOME (s |+ (v1,t2))
    ‖ (t1,Var v2) → if oc s t1 v2 then NONE else SOME (s |+ (v2,t1))
    ‖ (⟨t11, t12⟩,⟨t21, t22⟩) → do sx ← tunify s t11 t21; tunify sx t12 t22 od
    ‖ (Const c1,Const c2) → if c1 = c2 then SOME s else NONE
    ‖ _ → NONE
  else
    ARB
```

Figure 3.1: First-order unification: the `unify` function is the `then` branch of the `if`.

same variable, no binding is added, otherwise a binding from the variable to the other term is added, provided the variable doesn't occur in the term. The next pattern is for when both terms walk to pairs. In this case, we recursively try to unify the left sides of the pairs. If that unification succeeds, yielding an extended substitution sx, we then try to unify the right sides of the pairs in the context of sx. Finally, the penultimate pattern deals with two constants, which either add no bindings if equal, or lead to failure if unequal, and the last pattern is for failing on terms of different shapes.

3.4 Termination

Three termination conditions are generated by HOL4, corresponding to the need for a well-founded relation and the two recursive calls:

1. `WF` R

2. $\forall\, t_2\ t_1\ s\ t_{11}\ t_{12}\ t_{21}\ t_{22}.$
   $\texttt{wfs}\ s \wedge \texttt{walk}\ s\ t_1 = \langle t_{11}, t_{12}\rangle \wedge \texttt{walk}\ s\ t_2 = \langle t_{21}, t_{22}\rangle \Rightarrow$
   $R\ (s, t_{11}, t_{21})\ (s, t_1, t_2)$

3. $\forall\, t_2\ t_1\ s\ t_{11}\ t_{12}\ t_{21}\ t_{22}\ sx.$
   $\texttt{wfs}\ s \wedge (\texttt{walk}\ s\ t_1 = \langle t_{11}, t_{12}\rangle \wedge \texttt{walk}\ s\ t_2 = \langle t_{21}, t_{22}\rangle) \wedge$
   $\texttt{tunify_tupled_aux}\ R\ (s, t_{11}, t_{21}) = \texttt{SOME}\ sx \Rightarrow$
   $R\ (sx, t_{12}, t_{22})\ (s, t_1, t_2)$

A call `tunify_tupled_aux` R *args* is a guarded call to the only-partially defined `tunify`: any recursive calls must be on arguments that are R-smaller than *args*. The call appears in Condition 3 because the argument sx in the second recursive call `tunify` $sx\ t_{12}\ t_{22}$ is the result of the first recursive call. This is thus an instance of nested recursion.

The `unify` function walks the subterms being considered in the current substitution before case analysis. The key to the termination argument is that size of the subterms, considered in the context of the updated substitution, goes down on every recursive call. The termination relation $\texttt{unify}_{\texttt{TR}}$, defined below, makes this statement in the final conjunct. The other conjuncts are also satisfied by the algorithm and are required to ensure that $\texttt{unify}_{\texttt{TR}}$ is well-founded.

Definition 21. Termination relation for `unify`

$$\texttt{unify}_{\texttt{TR}}\ (sx, c_1, c_2)\ (s, t_1, t_2) \iff$$
$$\texttt{wfs}\ sx \wedge s \sqsubseteq sx \wedge \texttt{allvars}\ sx\ c_1\ c_2 \subseteq \texttt{allvars}\ s\ t_1\ t_2 \wedge$$
$$\texttt{measure}\ (\texttt{pair_count} \circ \texttt{walk}^*\ sx)\ c_1\ t_1$$

Theorem 2. $\texttt{unify}_{\texttt{TR}}$ *is well-founded*

$$\vdash \texttt{WF}\ \texttt{unify}_{\texttt{TR}}$$

Proof. By contradiction. If there is an infinite $\texttt{unify}_{\texttt{TR}}$-chain, then the set of variables in the arguments (`allvars`) must reach a fixpoint because each successive set is a subset of its predecessor, and the sets are finite. As the set of variables is getting smaller, the substitutions are allowed to get larger (the $\sqsubseteq$ relation). However, once the set of variables reaches its fixpoint, the substitutions will be drawing on a fixed source for new variable bindings, so they must also reach a fixpoint. Once the substitution (sx) is fixed, the first argument of the `measure` conjunct becomes fixed. Hence the supposedly infinite chain would have to stop (when $sx \triangleleft c_1$ has zero size): contradiction. □

We thereby satisfy Termination Condition 1. Condition 2 is easy because the substitution doesn't change.

Lemma 19. *Termination Condition 2*

$$\vdash \texttt{wfs}\ s \wedge \texttt{walk}\ s\ t_1 = \langle t_{11}, t_{12}\rangle \wedge \texttt{walk}\ s\ t_2 = \langle t_{21}, t_{22}\rangle \Rightarrow$$
$$\texttt{unify}_{\texttt{TR}}\ (s, t_{11}, t_{21})\ (s, t_1, t_2)$$

Proof. For the conjunct involving `allvars`: either $t_1 = \langle t_{11}, t_{12}\rangle$ or the pair is in the range of the substitution, and similarly for t_2. The other $\texttt{unify}_{\texttt{TR}}$ conjuncts are simple. □

Condition 3, however, requires some work. We define another relation, $\texttt{subst}_{\texttt{R}}$, weaker than $\texttt{unify}_{\texttt{TR}}$, which asserts that the variables of the result substitution all come from the arguments. The $\texttt{subst}_{\texttt{R}}$ relation serves as a bridge: weak enough that we can prove it is satisfied by `tunify` by induction and strong enough that it implies $\texttt{unify}_{\texttt{TR}}$. We use a relation that restricts the substitution only since at this point we can't say much about recursive calls without proving $\texttt{unify}_{\texttt{TR}}$ for each call.

Definition 22. Relation between the output substitution and input arguments

$$\texttt{subst}_{\texttt{R}}\ sx\ s\ t_1\ t_2 \iff$$
$$\texttt{wfs}\ sx \wedge s \sqsubseteq sx \wedge \texttt{substvars}\ sx \subseteq \texttt{allvars}\ s\ t_1\ t_2$$

We now establish the relationship between $\texttt{subst}_{\texttt{R}}$ and $\texttt{unify}_{\texttt{TR}}$.

Lemma 20. $\texttt{subst}_{\texttt{R}}$ *implies* $\texttt{unify}_{\texttt{TR}}$ *on subterms*

$$\begin{array}{l} \vdash \texttt{wfs}\ s \land \texttt{walk}\ s\ t_1 = \langle t_{11}, t_{12} \rangle \land \texttt{walk}\ s\ t_2 = \langle t_{21}, t_{22} \rangle \land \\ \quad (\texttt{subst}_{\texttt{R}}\ sx\ s\ t_{11}\ t_{21} \lor \texttt{subst}_{\texttt{R}}\ sx\ s\ t_{12}\ t_{22}) \Rightarrow \\ \qquad \texttt{unify}_{\texttt{TR}}\ (sx, t_{12}, t_{22})\ (s, t_1, t_2) \end{array}$$

Proof. The `allvars` conjunct uses the fact that the variables in the result of a walk are all in the range of the substitution.

For the `measure` conjunct we observe that $\texttt{walk}\ sx\ t_1 = \texttt{walk}\ s\ t_1$ because s is a submap of sx and $\texttt{walk}\ s\ t_1$ isn't a variable. It is then clear that $sx \lhd t_{12}$ is a subterm of $sx \lhd t_1$. □

Lemma 21. `unify` *implies* $\texttt{subst}_{\texttt{R}}$

$$\begin{array}{l} \vdash \texttt{wfs}\ s \land \texttt{tunify_tupled_aux}\ \texttt{unify}_{\texttt{TR}}\ (s, t_1, t_2) = \texttt{SOME}\ sx \Rightarrow \\ \qquad \texttt{subst}_{\texttt{R}}\ sx\ s\ t_1\ t_2 \end{array}$$

Proof. By well-founded induction (knowing that $\texttt{unify}_{\texttt{TR}}$ is well-founded). □

Finally we can prove the outstanding termination condition.

Lemma 22. *Termination Condition 3*

$$\begin{array}{l} \vdash \texttt{wfs}\ s \land \texttt{walk}\ s\ t_1 = \langle t_{11}, t_{12} \rangle \land \texttt{walk}\ s\ t_2 = \langle t_{21}, t_{22} \rangle \land \\ \quad \texttt{tunify_tupled_aux}\ \texttt{unify}_{\texttt{TR}}\ (s, t_{11}, t_{21}) = \texttt{SOME}\ sx \Rightarrow \\ \qquad \texttt{unify}_{\texttt{TR}}\ (sx, t_{12}, t_{22})\ (s, t_1, t_2) \end{array}$$

Proof. From the lemmas above. □

3.5 Correctness

There are three parts to the correctness statement:

1. (soundness) if `unify` succeeds then its result is a (well-formed) unifier;
2. (generality) if `unify` succeeds then its result is most general; and
3. (completeness) if there exists a unifier of $s \lhd t_1$ and $s \lhd t_2$, then $\texttt{unify}\ s\ t_1\ t_2$ succeeds.

A substitution s is a *unifier* of terms t_1 and t_2 if it is well-formed and $s \lhd t_1 = s \lhd t_2$. It is not generally true that the result of `unify` is idempotent. But `unify` preserves well-formedness, which (as per Theorem 1) ensures the well-formed result can be collapsed into an idempotent substitution.

3.5.1 Soundness

Theorem 3. *The result of* `unify` *is a unifier and a well-formed extension*

$$\vdash \texttt{wfs}\ s\ \wedge\ \texttt{unify}\ s\ t_1\ t_2\ =\ \texttt{SOME}\ sx\ \Rightarrow$$
$$\texttt{wfs}\ sx\ \wedge\ s\ \sqsubseteq\ sx\ \wedge\ sx\ \triangleleft\ t_1\ =\ sx\ \triangleleft\ t_2$$

Proof. The first two conjuncts, that s is a submap of sx and sx is well-formed, are corollaries of Lemma 21. Essentially, `unify` only updates the substitution, and then only with variables that aren't already in the domain.

The rest follows by recursion induction on `unify`, using Lemma 9. □

Given Lemma 9 and Theorem 3, we can equally regard `unify` $s\ t_1\ t_2$ as calculating a unifier for t_1 and t_2 or for the terms-in-context $s \triangleleft t_1$ and $s \triangleleft t_2$. It is not always the case, however, that `unify` $s\ t_1\ t_2$ = `unify` s ($s \triangleleft t_1$) ($s \triangleleft t_2$). The latter result may be "less triangular" than the former. For example, take t_1 = `Var` x and t_2 = `Var` y in the substitution $s = \{x \mapsto \langle \texttt{Var}\ z, \texttt{Var}\ z\rangle, z \mapsto \langle c_1, c_2\rangle\}$ where c_1 and c_2 are ground terms. Calling `unify` on these terms results in a substitution binding y to $\langle \texttt{Var}\ z, \texttt{Var}\ z\rangle$, but if the substitution is applied before calling `unify` then y is bound to $\langle\langle c_1, c_2\rangle, \langle c_1, c_2\rangle\rangle$.

3.5.2 Generality

The context provided by the input substitution is relevant to our notion of a most general unifier, which differs from the usual context-free notion. A unifier of terms in context is *most general* if it can be composed with another substitution to equal any other unifier *in the same context.* In the empty context, however, the notions of most general unifier coincide.

Lemma 23. *The kinds of extensions made by unify are innocuous*

$$\vdash \texttt{wfs}\ s_1\ \wedge\ \texttt{wfs}\ (s\ \texttt{|+}\ (vx\texttt{,}tx))\ \wedge\ vx\ \notin\ \texttt{FDOM}\ s\ \wedge$$
$$s_1\ \triangleleft\ \texttt{Var}\ vx\ =\ s_1\ \triangleleft\ s\ \triangleleft\ tx\ \Rightarrow$$
$$\forall t.\ s_1\ \triangleleft\ (s\ \texttt{|+}\ (vx\texttt{,}tx))\ \triangleleft\ t\ =\ s_1\ \triangleleft\ s\ \triangleleft\ t$$

Proof. By recursion induction on `walk*`. □

Lemma 24. *The result of* `unify` *is most general (in context)*

$$\vdash \texttt{wfs}\ s\ \wedge\ \texttt{unify}\ s\ t_1\ t_2\ =\ \texttt{SOME}\ sx\ \wedge\ \texttt{wfs}\ s_2\ \wedge$$
$$s_2\ \triangleleft\ s\ \triangleleft\ t_1\ =\ s_2\ \triangleleft\ s\ \triangleleft\ t_2\ \Rightarrow$$
$$\forall t.\ s_2\ \triangleleft\ sx\ \triangleleft\ t\ =\ s_2\ \triangleleft\ s\ \triangleleft\ t$$

Proof. By recursion induction on `unify` using the lemma above. □

Theorem 4. *The result of* `unify` *is most general (empty context)*

$$\vdash \texttt{unify FEMPTY}\ t_1\ t_2\ =\ \texttt{SOME}\ sx\ \Rightarrow$$
$$\forall s.\ \texttt{wfs}\ s\ \wedge\ s\ \triangleleft\ t_1\ =\ s\ \triangleleft\ t_2\ \Rightarrow\ \exists s'.\ \forall t.\ s'\ \triangleleft\ sx\ \triangleleft\ t\ =\ s\ \triangleleft\ t$$

Remark 1. By the lemma above we see that the witness is s itself.

3.5.3 Completeness

We now turn to the completeness result, for which we use the following lemma.

Lemma 25. *A variable and a term containing that variable remain different under application*

$$\vdash \texttt{oc}\ s\ t\ v \land (\forall w.\ t \neq \texttt{Var}\ w) \land \texttt{wfs}\ s \land \texttt{wfs}\ s_2 \Rightarrow$$
$$s_2 \lhd \texttt{Var}\ v \neq s_2 \lhd s \lhd t$$

Proof. By considering the term sizes. (Section 1.4.3 gives the proof tactic in detail.) □

Theorem 5. *If the terms are unifiable, then* `unify` *succeeds*

$$\vdash \texttt{wfs}\ s \land \texttt{wfs}\ s_2 \land s_2 \lhd s \lhd t_1 = s_2 \lhd s \lhd t_2 \Rightarrow$$
$$\exists\, sx.\ \texttt{unify}\ s\ t_1\ t_2 = \texttt{SOME}\ sx$$

Proof. By recursion induction on `unify`, using Lemma 25 for the non-trivial occurs checks, and using Lemma 24 for the recursive case. □

3.6 Other algorithms

Unification is a well-studied problem, and there are a number of different approaches to solving it. The algorithm we have seen can be described as an accumulator-passing version of the recursive descent approach. In this section we will look at a recursive descent algorithm that isn't accumulator-passing, and returns idempotent rather than triangular unifiers. We will compare this idempotent algorithm to the one in Section 3.3, and see that the termination argument uses a different well-founded relation. We will also look at the transformation system given in [Martelli and Montanari 1982] to describe unification algorithms, also designed for those with an idempotent output.

Definition 23. Unification with idempotent substitutions

```
Iunify t1 t2 =
  case (t1,t2) of
    (Var v1,Var v2) →
      SOME (if v1 = v2 then FEMPTY else FEMPTY |+ (v1,Var v2))
  || (Var v1,t2) →
      if v1 ∈ vars t2 then NONE else SOME (FEMPTY |+ (v1,t2))
  || (t1,Var v2) →
      if v2 ∈ vars t1 then NONE else SOME (FEMPTY |+ (v2,t1))
  || (⟨t11, t12⟩,⟨t21, t22⟩) →
      do
        s1 ← Iunify t11 t21;
        s2 ← Iunify (s1 " t12) (s1 " t22);
        SOME (s2 ⊙ s1)
      od
  || (Const c1,Const c2) → if c1 = c2 then SOME FEMPTY else NONE
  || _ → NONE
```

Definition 23 gives a unification algorithm similar to the one we have seen except that it maintains idempotent substitutions. This algorithm was first described in [Manna and Waldinger 1981]; the version here is adapted from [Slind 1999]. The first difference to note is that `Iunify` doesn't take an input substitution. The call analogous to `unify` s t_1 t_2 would be `Iunify (`s `"` t_1`) (`s `"` t_2`)` if s is idempotent, otherwise `Iunify (`$s \lhd t_1$`) (`$s \lhd t_2$`)`. The occurs check in `Iunify` (*e.g.* $v_1 \in$ `vars` t_2) also takes no substitution for context.

In the pair case, the substitution obtained by unifying the left sides of the pairs is applied to the right sides before unifying them. This is still an instance of nested recursion, since the result of one recursive call determines the arguments to the other recursive call. The result of the second recursive call, s_2, is not necessarily an extension of s_1, so information from both is required in the final result and the two substitutions are composed. The substitution called sx in our algorithm corresponds to s_1 here; we don't name the substitution corresponding to s_2.

Proofs of termination and correctness of `Iunify` are provided by [Paulson 1985] and [Slind 1999]. The termination relation is a lexicographical combination: either the variables in one pair of arguments are a proper subset of the variables in the other, or they have the same variables but the terms are smaller. We can write this in HOL as

```
inv_image
  ((λ s′ s. FINITE s ∧ s′ ⊂ s) LEX measure pair_count LEX
   measure pair_count) (λ (t1,t2). (vars t1 ∪ vars t2,t1,t2))
```

The first recursive call of `Iunify` is on the left hand sides of the pairs, so the arguments will always be smaller terms with no more variables than those of the original call. The second recursive call might be on larger terms, since applying s_1 could send some variable to a larger term. But in that case, applying s_1 eliminates a variable, so the variables in the arguments of the second recursive call are a proper subset of those of the original call. This argument corresponds to our Lemmas 19 and 22.

The well-foundedness of the lexicographic combination, which corresponding to Theorem 2, is much simpler than our theorem, since it follows directly from the well-foundedness of `measure` and finite proper subset, and the well-foundedness preservation of `inv_image` and `(LEX)`. Using a lexicographic combination often makes proving well-foundedness easy; we saw a similar proof in `walk*`'s termination argument, for Lemma 7. Our termination relation, $\texttt{unify}_{\texttt{TR}}$, is not a lexicographic combination. It may, however, be possible to express $\texttt{unify}_{\texttt{TR}}$ as a subrelation of a well-founded lexicographic combination. We haven't worked out how yet, but it would make the well-foundedness proof simpler if it's possible.

Now we shall look at the non-deterministic algorithm for unification given in [Martelli and Montanari 1982] in the form of a transformation system. The system, which we will call the Martelli-Montanari transformation system (MMTS), is notable as a framework in which many unification algorithms can be expressed as particular choices of the order in which to take transformations. A transformation system, or rewriting system, is a set of rules to transform a problem that can be applied in any order whenever applicable. A unification problem in the MMTS is represented as a set of equations,

initially $\{t_1 =_? t_2\}$ where these are the input terms. The rules, adapted from [Baader and Nipkow 1998], are as follows.

Delete Remove any equation $t =_? t$.

Orient Replace $t =_? \texttt{Var}\ x$ by $\texttt{Var}\ x =_? t$, as long as t is not a variable.

Decompose Replace $\langle t_{11}, t_{12}\rangle =_? \langle t_{21}, t_{22}\rangle$ by $t_{11} =_? t_{21}$ and $t_{12} =_? t_{22}$.

Eliminate If $\texttt{Var}\ x =_? t$ is in the problem, and $x \notin \texttt{vars}\ t$, and x occurs in another equation in the problem, then apply the substitution $\{x \mapsto t\}$ to both sides of each equation in the rest of the problem. Keep $\texttt{Var}\ x =_? t$.

When no more rules can be applied then if the original terms were unifiable, the final set of equations will all be of the form $\texttt{Var}\ x =_? t$ which can be read as bindings of an idempotent unifying substitution.

Termination of the MMTS means that for any initial set of equations, eventually no more rules will be applicable. This is proved by defining a well-founded relation that relates the sets of equations before and after application of any rule. The relation is a lexicographic combination of the following three measures:

- n_1, the number of variables that appear in an equation of any form other than $\texttt{Var}\ x =_? t$;
- n_2, the total size of the problem, given by $\sum_{t_1 =_? t_2} \texttt{pair_count}\ t_1\ \texttt{+}\ \texttt{pair_count}\ t_2$; and
- n_3, the number of equations of the form $t =_? \texttt{Var}\ x$ or $t =_? t$.

All the rules satisfy this lexicographic combination. The Delete and Orient rules both decrease n_3 and cannot increase n_1 or n_2. The Decompose rule decreases n_2 and cannot increase n_1. The Eliminate rule decreases n_1, since applying the substitution eliminates the variable from all other equations, and the rule only applies when there is an equation containing the variable.

The worst-case time complexity of recursive descent unification algorithms is exponential in the size of the input terms. Consider unifying $\langle x, y\rangle$ and $\langle\langle z, z\rangle, \langle x, x\rangle\rangle$. The first binding made by both `unify` and `Iunify` is $x \mapsto \langle z, z\rangle$. Then `Iunify` substitutes this through the second half of the pair, and `unify` effectively does the same thing since the substitution binding x will be an argument in the recursion. Thus we have to unify y with $\langle\langle z, z\rangle, \langle z, z\rangle\rangle$, and the size of the second half of the second pair has doubled. It's not hard to see that a linear increase in the size of the input terms can lead to an exponential increase in effective size through such doubling. However, unifying $\langle y, x\rangle$ with $\langle\langle x, x\rangle, \langle z, z\rangle\rangle$, which is essentially the same problem, avoids the exponential increase in `unify` since the binding for x never needs to be used. The MMTS can also exhibit exponential behavior. The issue, as with the other algorithms, is the order in which unification subproblems are tackled.

The fastest unification algorithms are linear time. To achieve this complexity, a number of techniques have been developed, including

- spending time choosing the next subproblem carefully, exploiting an ordering on the variables that minimises changes to other subproblems when a new binding is added;

- grouping equated terms into equivalence classes, also known as using "multi-equations"; and

- using efficient representations of terms as graphs, with complicated use of pointers.

Examples of efficient algorithms using these techniques can be found in [Martelli and Montanari 1982] and [Paterson and Wegman 1978].

Chapter 4

Nominal Unification

4.1 Introduction

The main question in nominal unification is the same as in the first-order case: "Can these terms be equal?". But the terms are now nominal terms, and equality between them isn't syntactic identicality.

Nominal logic [Pitts 2003] is a version of first-order logic designed for modelling *names* and *binding*. Binding constructs are prevalent in the syntax of programming languages and other formalisms. A primary example is lambda abstractions, for example $\lambda\, x\ \ y.$ `(`x `+` y`) *` z where the names x and y are bound by the lambda, although z is free. Other examples from mathematics include syntax for integrals $\int_0^\infty g(a)\mathrm{d}a$, which here binds the name a, and products $\prod_{b=1}^{10} b^2$, here binding b. In each case the bound item is a "dummy variable" whose name should not matter, thus we would like to say $\lambda\, x\ \ y.$ `(`x `+` y`) *` z and $\lambda\, a\ \ b.$ `(`a `+` b`) *` z are syntactically equivalent. This notion of equivalence is called α-equivalence. (Assuming the mathematical operators have their usual meanings, our example function is extensionally equivalent to $\lambda\, x\ \ y.$ x `*` z `+` y `*` z. But α-equivalence is a syntactic property only, and we don't consider any equivalences more complicated than renaming bound variables.)

Working with α-equivalence is complicated by the fact that names can inadvertently collide. We could not rename any bound variable to z in the example above: $\lambda\, x\ \ z.$ `(`x `+` z`) *` z is a different function from $\lambda\, x\ \ y.$ `(`x `+` y`) *` z. A nominal unification algorithm must deal with this issue, and nominal logic provides the machinery to do so.

The syntax of nominal terms allows us to express binding constructs generically. A construct like $\lambda\, x.$ x `+ 1` might be represented by the nominal term

$\langle$`Const`$_\texttt{n}$ `"lambda"`, `Tie "x"` $\langle$`Const`$_\texttt{n}$ `"+"`, $\langle$`Nom "x"`, `Const`$_\texttt{n}$ `"1"`$\rangle_\texttt{n}\rangle_\texttt{n}\rangle_\texttt{n}$

Binding of a name is represented by the nominal term constructor `Tie` ("a tie" is another name for a binder), and the name itself is represented by a `Nom` term. As in first-order terms, we can also have variables in nominal terms, which usually stand for a piece of syntax. Thus $\langle$`Const`$_\texttt{n}$ `"lambda"`, `Tie "a" (Sus []` X`)`$\rangle_\texttt{n}$ might represent $\lambda\, a.$ X where X is a metavariable standing for the body of the lambda abstraction. Nominal variables are called suspensions, hence the `Sus` constructor. We will use capital letters to distinguish metavariables (suspensions) from variables in the syntax (`Nom`s).

Consider unification of nominal terms representing $\lambda\, a.\ X$ and $\lambda\, b.\ b$. A reasonable answer is the substitution $\{X \mapsto$ `Nom` $a\}$, since a plays the role of the bound variable where X appears, corresponding to b in the other term. Similarly, a substitution unifying $\lambda\, a.\ X$ and $\lambda\, b.\ c$ is $\{X \mapsto$ `Nom` $c\}$, since in this case the c would be free in both terms. To unify $\lambda\, a.\ X$ and $\lambda\, b.\ Y$, however, we need two new nominal concepts: permutations and freshness constraints. Informally, the answer to the unification problem is that X should be bound to the same term as Y, but with all the b's replaced by a's. Furthermore, no a's should appear free in Y, since they would be inadvertently captured when put in place of X.

Replacement of one name by another is done in nominal logic with *permutations*. A permutation is a bijection on the set of names, so is never a one-way replacement: if some name a is replaced by b, then b will be replaced by something else. Permutations are made out of swaps (transpositions); the swap `(`a`,`b`)` sends a to b and b to a. We can now write the substitution part of a unifier of $\lambda\, a.\ X$ and $\lambda\, b.\ Y$ as $\{X \mapsto$ `Sus [(`a`,`b`)]` $Y\}$. This substitution binds X to a suspension representing Y with the names a and b permuted.

The second part of the unifier is the *freshness constraint* that no a's appear free in Y, and this is written $a \# Y$. This constraint is necessary because binding Y to `Nom` a, for example, would yield $\lambda\, a.\ b$ and $\lambda\, b.\ a$, which are not α-equivalent. A complete nominal unifier consists of a substitution and set of freshness constraints, called a *freshness environment*. The freshness environment provides a context for equality between nominal terms, which should capture α-equivalence. We write $fe \vdash t_1 \approx t_2$ to mean that two terms are equivalent with respect to a freshness environment. For example, we have

$$\{(a,Y)\} \vdash \texttt{Tie}\ a\ (\texttt{Sus []}\ X) \approx \texttt{Tie}\ b\ (\texttt{Sus [(}a\texttt{,}b\texttt{)]}\ Y)$$

which shows that the unifier we constructed for $\lambda\, a.\ X$ and $\lambda\, b.\ Y$ (encoded simply as ties) does make them equivalent.

We can do logic programming with nominal logic, an idea first developed by James Cheney in his doctoral dissertation [Cheney 2004]. One of the first nominal logic programming systems was αProlog [Cheney and Urban 2004], which inspired a nominal version of miniKanren called αKanren [Byrd and Friedman 2007]. In addition to the new term constructors, for names and ties, αKanren includes a new relation-like procedure for asserting a freshness constraint. And of course the $\equiv$ relation is used for nominal rather than first-order unification. Nominal unification was first defined in [Urban et al. 2004], as a non-deterministic algorithm working with idempotent substitutions. We formalise an accumulator-passing deterministic variant of this algorithm, that uses triangular substitutions, as implemented in αKanren. More information on nominal logic programming can be found in [Cheney and Urban 2008].

In the next section we look at nominal terms and substitutions in more detail, generalising the first-order theory from Chapter 2 as a foundation for nominal unification. We then define a nominal unification algorithm that works with triangular substitutions, and prove the algorithm correct. The end of the chapter includes the results of

experiments comparing the efficiency of the triangular algorithm to one that maintains idempotent substitutions.

4.2 Nominal terms

Nominal terms extend first-order terms with two new constructors, one for names (also called atoms), and one for *ties*, which represent binders (terms with a bound name). We also replace the `Var` constructor with a constructor for *suspensions*, the nominal analogue of variables. A suspension is made up of a variable name and a permutation of names, and stands for the variable after application of the permutation. When (if) the variable is bound, the permutation can be applied further.

Definition 24. Concrete nominal terms

```
α Cterm
  = CNom of string
  | CSus of (string, string) alist => num
  | CTie of string => α Cterm
  | CPair_n of α Cterm => α Cterm
  | CConst_n of α
```

We represent permutations as lists of pairs of names; such a list stands for an ordered composition of swaps, with the head of list applied last. There may be more than one list representing the same permutation. For example `[(a,b)]` and `[(b,d); (a,d); (c,c); (d,b)]` represent the same permutation, if we assume the four names are distinct. To see the effect of the second permutation on d, start with the swap `(d,b)`, which sends d to b, then the swap `(c,c)` does nothing, the swap `(a,d)` does nothing to b, and finally the swap `(b,d)` sends our b back to d.

We abstract over different lists representing the same permutation by creating a *quotient type*. The nominal term data type α `nterm` is the quotient of the concrete type above by permutation equivalence `(==)`. The terms `Sus [(a,b)]` x and `Sus [(c,c); (b,a)]` x of this type are equal because the permutations are equivalent. Constructors for α `nterms` are the same as for α `Cterms` but with the `C` prefix removed (and we write pairs with angle brackets).

Application of a permutation π to a name a is written $\pi \bullet a$. Permutations are composed by appending the lists of swaps, and are inverted by reversing the list; when the lists represents a permutation we write p^{-1} instead of `REVERSE` p. In other words, we have the following theorems, which we won't prove here:

$$\vdash (p_1 \texttt{ ++ } p_2) \bullet s = p_1 \bullet p_2 \bullet s$$
$$\vdash p^{-1} \bullet p \bullet s = s \wedge p \bullet p^{-1} \bullet s = s$$

We lift permutation application to terms in the following definition.

Definition 25. Applying a permutation to a term

$$\pi \bullet \texttt{Nom}\ a = \texttt{Nom}\ (\pi \bullet a)$$
$$\pi \bullet \texttt{Sus}\ p\ v = \texttt{Sus}\ (\pi \ \texttt{++}\ p)\ v$$
$$\pi \bullet \texttt{Tie}\ a\ t = \texttt{Tie}\ (\pi \bullet a)\ (\pi \bullet t)$$
$$\pi \bullet \langle t_1, t_2 \rangle_{\texttt{n}} = \langle \pi \bullet t_1, \pi \bullet t_2 \rangle_{\texttt{n}}$$
$$\pi \bullet \texttt{Const}_{\texttt{n}}\ c = \texttt{Const}_{\texttt{n}}\ c$$

Following the example of the first-order algorithm, we define the "walk" operation that finds a suspension's ultimate binding. The termination argument for $\texttt{vwalk}_{\texttt{n}}$ is the same as in the first-order case; the permutation doesn't play a part in the recursion.

Definition 26. Walking a suspension

```
wfs_n s ⇒
  vwalk_n s π v =
    case s ' v of
      SOME (Sus p u) → vwalk_n s (π ++ p) u
    || SOME t → π • t
    || NONE → Sus π v
```

The role of the permutation argument of $\texttt{vwalk}_{\texttt{n}}$ is to accumulate the permutations on suspensions in the range of the substitution. The initial call to $\texttt{vwalk}_{\texttt{n}}$ will usually take the empty permutation, since this is provided by $\texttt{walk}_{\texttt{n}}$. The behaviour of the accumulator is summarised in the following lemma.

Lemma 26. *The initial permutation for a walk can be applied at the end*

$$\vdash \texttt{wfs}_{\texttt{n}}\ s \Rightarrow \texttt{vwalk}_{\texttt{n}}\ s\ \pi\ v = \pi \bullet \texttt{vwalk}_{\texttt{n}}\ s\ \texttt{[]}\ v$$

Proof. By recursion induction on $\texttt{vwalk}_{\texttt{n}}$. □

Nominal substitution application is analogous to the first-order case: $\texttt{walk}_{\texttt{n}}$ calls $\texttt{vwalk}_{\texttt{n}}\ s\ p\ v$ for a suspension $\texttt{Sus}\ p\ v$, otherwise returns its argument; $\texttt{walk}^*_{\texttt{n}}$ uses $\texttt{walk}_{\texttt{n}}$, recurring on ties as well as pairs.

Lemma 27. *Nominal substitution application*

```
⊢ wfs_n s ⇒
    s ◁_n t =
      case walk_n s t of
        Tie a t → Tie a (s ◁_n t)
      || ⟨t1, t2⟩_n → ⟨s ◁_n t1, s ◁_n t2⟩_n
      || t → t
```

Freshness of a name for a term, and equivalence between terms are defined as in [Urban et al. 2004]. We define freshness as a function but equivalence as an inductive relation; the name of the relation is `equiv`. The disagreement set of two permutations is the set of names which aren't sent to the same name by each permutation.

Definition 27. Disagreement sets

$$\texttt{dis_set}\ \pi_1\ \pi_2\ =\ \{a\ \mid\ \pi_1 \bullet a\ \neq\ \pi_2 \bullet a\}$$

Definition 28. Freshness of a name for a term

$$\begin{array}{l} fe\ \vdash\ a\ \#\ \texttt{Nom}\ b\ \iff\ a\ \neq\ b \\ fe\ \vdash\ a\ \#\ \texttt{Sus}\ \pi\ v\ \iff\ (\pi^{-1} \bullet a, v)\ \in\ fe \\ fe\ \vdash\ a\ \#\ \texttt{Tie}\ b\ t\ \iff\ a\ =\ b\ \vee\ a\ \neq\ b\ \wedge\ fe\ \vdash\ a\ \#\ t \\ fe\ \vdash\ a\ \#\ \langle t_1, t_2\rangle_{\texttt{n}}\ \iff\ fe\ \vdash\ a\ \#\ t_1\ \wedge\ fe\ \vdash\ a\ \#\ t_2 \\ fe\ \vdash\ a\ \#\ \texttt{Const}_{\texttt{n}}\ c\ \iff\ \texttt{T} \end{array}$$

Definition 29. Equivalence between terms

$$\frac{}{fe\ \vdash\ \texttt{Nom}\ a\ \approx\ \texttt{Nom}\ a}$$

$$\frac{\forall a.\ a\ \in\ \texttt{dis_set}\ p_1\ p_2\ \Rightarrow\ (a, v)\ \in\ fe}{fe\ \vdash\ \texttt{Sus}\ p_1\ v\ \approx\ \texttt{Sus}\ p_2\ v}$$

$$\frac{fe\ \vdash\ t_1\ \approx\ t_2}{fe\ \vdash\ \texttt{Tie}\ a\ t_1\ \approx\ \texttt{Tie}\ a\ t_2}$$

$$\frac{a_1\ \neq\ a_2 \quad fe\ \vdash\ a_1\ \#\ t_2 \quad fe\ \vdash\ t_1\ \approx\ [(a_1, a_2)] \bullet t_2}{fe\ \vdash\ \texttt{Tie}\ a_1\ t_1\ \approx\ \texttt{Tie}\ a_2\ t_2}$$

$$\frac{fe\ \vdash\ t_{11}\ \approx\ t_{21} \quad fe\ \vdash\ t_{12}\ \approx\ t_{22}}{fe\ \vdash\ \langle t_{11}, t_{12}\rangle_{\texttt{n}}\ \approx\ \langle t_{21}, t_{22}\rangle_{\texttt{n}}}$$

Lemma 28. `equiv` *is an equivalence relation*

$$\begin{array}{l} fe\ \vdash\ t\ \approx\ t \\ fe\ \vdash\ t_1\ \approx\ t_2\ \Rightarrow\ fe\ \vdash\ t_2\ \approx\ t_1 \\ fe\ \vdash\ t_1\ \approx\ t_2\ \wedge\ fe\ \vdash\ t_2\ \approx\ t_3\ \Rightarrow\ fe\ \vdash\ t_1\ \approx\ t_3 \end{array}$$

Proof. For transitivity (the other results are trivial), we first establish

$$fe\ \vdash\ t_1\ \approx\ t_2\ \wedge\ fe\ \vdash\ t_2\ \approx\ \pi \bullet t_2\ \Rightarrow\ fe\ \vdash\ t_1\ \approx\ \pi \bullet t_2$$

by rule induction on `equiv` (on the derivation of $fe\ \vdash\ t_1\ \approx\ t_2$). Another rule induction then establishes the final result, without needing an ugly induction on term sizes, as in [Urban et al. 2004]. □

4.3 Definition

In the first phase of nominal unification (as defined in [Urban et al. 2004]), a substitution is constructed along with a freshness environment. The substitution is akin to the substitution generated in first-order unification, and the freshness environment represents any constraints picked up due to unification of ties. We represent freshness constraints by pairs, *i.e.*, (a, X) represents $a \# X$.

The second phase of unification checks to see if the freshness constraints are consistent, possibly dropping irrelevant constraints along the way. A constraint such as $a \# X$ can become irrelevant when the first phase adds a binding from X to $\texttt{Const}_\texttt{n}\ 3$, for example, because a cannot appear free in the constant. Conversely the freshness environment can become inconsistent when a constraint like $a \# X$ is violated by a binding from X to, say, $\langle \texttt{Nom}\ a, \texttt{Const}_\texttt{n}\ 2\rangle_\texttt{n}$. If the check succeeds, the substitution and the new freshness environment, which together form a nominal unifier, are returned.

The use of triangular substitutions and the accumulator-passing style means that our definition of nominal unification differs from [Urban et al. 2004]: the substitution returned from the first phase is referred to as the freshness constraints are checked in the second phase.

Definition 32 (Figure 4.1 on page 54) is our definition of `nomunify` in HOL. In both phases, we use the auxiliary `term_fcs` (Definition 30). This function is given a name and a term, and constructs a minimal freshness environment sufficient to ensure that the name is fresh for the term. If this is impossible (*i.e.*, if the name is free in the term), `term_fcs` returns `NONE`.

Definition 30. Constructing a minimal freshness environment for a freshness constraint

```
term_fcs a t =
  case t of
    Nom b → if a = b then NONE else SOME { }
  ‖ Sus p v → SOME { (p⁻¹ • a, v) }
  ‖ Tie b′ tt → if a = b′ then SOME { } else term_fcs a tt
  ‖ ⟨t1, t2⟩n →
      do
        fe1 ← term_fcs a t1;
        fe2 ← term_fcs a t2;
        SOME (fe1 ∪ fe2)
      od
  ‖ Constn v8 → SOME { }
```

We write $fe \vdash a \# t$ to mean that a name is fresh for a term with respect to a freshness environment. The next few lemmas assert that our definition of `term_fcs` is correct. They are all proved by nominal term induction.

Lemma 29. *The result of* `term_fcs` *satisfies the freshness constraint*

$\vdash \texttt{term_fcs}\ a\ t = \texttt{SOME}\ fe \Rightarrow fe \vdash a \# t$

Lemma 30. *The result of* `term_fcs` *is minimal*

$\vdash \texttt{term_fcs}\ a\ t = \texttt{SOME}\ fe \wedge fe' \vdash a \# t \Rightarrow fe \subseteq fe'$

Lemma 31. *If* `term_fcs` *fails, the freshness constraint is unsatisfiable*

$\vdash \texttt{term_fcs}\ a\ t = \texttt{NONE} \Rightarrow \forall fe.\ fe \nvdash a \# t$

Following our strategy in the first-order case, $\texttt{unify}_\texttt{n}$ is defined *via* a total function $\texttt{tunify}_\texttt{n}$. The pair and constant cases are unchanged, and names are treated as constants.

With suspensions, there is an extra case to consider: if the variables are the same, we augment the freshness environment with a constraint $(a, s \lhd_\texttt{n} \texttt{Sus []}\ v)$ for every name a in the disagreement set of the permutations (done by `unify_eq_vars`). In the other suspension cases, we apply the inverse of the suspension's permutation to the term before performing the binding (done in `add_bdg`). (We invert the permutation so that applying the permutation to the term to which the variable is bound results in the term with which the suspension is supposed to unify.)

In the `Tie` case, a simple recursive descent is possible when the bound names are the same. Otherwise, we ensure that the first name is fresh for the body of the second term, and swap the two names in the second term before recursing.

Phase 2 is implemented by `verify_fcs`, which calls

$$\texttt{term_fcs}\ a\ (s \lhd_\texttt{n} \texttt{Sus []}\ v)$$

for each constraint (a, v) in the environment, accumulating the result.

Termination The termination argument for Phase 1 is analogous to the termination argument for `unify` in the first-order case. We use the same termination relation (this time measuring nominal term size, and ignoring the freshness environment). The extra termination condition for recursion down a `Tie` is handled like the easier of the `Pair` conditions because the substitution doesn't change and the freshness environment is irrelevant to termination.

Termination for Phase 2 depends only on the freshness environment being finite. We assume the freshness environment is finite in all valid inputs to `nomunify`, and it's easy to show that `term_fcs` (and hence Phase 1) preserves finiteness by structural induction on the nominal term.

4.4 Correctness

Just as in the first-order case, there are three parts to the correctness of a nominal unification algorithm, soundness, generality, and completeness. We follow similar strategies for proving these results, generalising the statements of lemmas in Section 3.5 where possible. The proofs are more complicated, however, mostly because we have to consider freshness environments, which are accumulated alongside substitutions. We develop results about the interaction between growing substitutions and growing freshness environments in order to carry out inductive proofs on $\texttt{unify}_\texttt{n}$.

We first state a couple of lemmas from [Urban et al. 2004] (original numbers in parentheses) that we reproved in HOL4 and will use in the proceeding sections.

Lemma 32. *Relating freshness to permutation application (2.7)*

$$\begin{aligned}
\vdash\ & (\forall t\ a\ \pi\ \mathit{fcs}.\ \mathit{fcs} \vdash a \ \#\ \pi \bullet t \Rightarrow \mathit{fcs} \vdash \pi^{-1} \bullet a \ \#\ t)\ \wedge \\
& (\forall t\ a\ \pi\ \mathit{fcs}.\ \mathit{fcs} \vdash \pi \bullet a \ \#\ t \Rightarrow \mathit{fcs} \vdash a \ \#\ \pi^{-1} \bullet t)\ \wedge \\
& \forall t\ a\ \pi\ \mathit{fcs}.\ \mathit{fcs} \vdash a \ \#\ t \Rightarrow \mathit{fcs} \vdash \pi \bullet a \ \#\ \pi \bullet t
\end{aligned}$$

Definition 31. Phase 1 (total version)

```
add_bdg π v t₀ (s,fe) =
  (let t = π⁻¹ • t₀ in
     if ocₙ s t v then NONE else SOME (s |+ (v,t),fe))

tunifyₙ (s,fe) t₁ t₂ =
  if wfsₙ s then
    case (walkₙ s t₁,walkₙ s t₂) of
      (Nom a₁,Nom a₂) → if a₁ = a₂ then SOME (s,fe) else NONE
    || (Sus π₁ v₁,Sus π₂ v₂) →
        if v₁ = v₂ then
          unify_eq_vars (dis_set π₁ π₂) v₁ (s,fe)
        else
          add_bdg π₁ v₁ (Sus π₂ v₂) (s,fe)
    || (Sus π₁ v₁,t₂) → add_bdg π₁ v₁ t₂ (s,fe)
    || (t₁,Sus π₂ v₂) → add_bdg π₂ v₂ t₁ (s,fe)
    || (Tie a₁ t₁,Tie a₂ t₂) →
        if a₁ = a₂ then
          tunifyₙ (s,fe) t₁ t₂
        else
          do
            fcs ← term_fcs a₁ (s ◁ₙ t₂);
            tunifyₙ (s,fe ∪ fcs) t₁ ([(a₁,a₂)] • t₂)
          od
    || (⟨t₁₁, t₁₂⟩ₙ,⟨t₂₁, t₂₂⟩ₙ) →
        do
          (sx,fex) ← tunifyₙ (s,fe) t₁₁ t₂₁;
          tunifyₙ (sx,fex) t₁₂ t₂₂
        od
    || (Constₙ c₁,Constₙ c₂) → if c₁ = c₂ then SOME (s,fe) else NONE
    || _ → NONE
  else
    ARB
```

Definition 32. Nominal unification in two phases

```
nomunify (s,fe) t₁ t₂ =
  do
    (sx,feu) ← unifyₙ (s,fe) t₁ t₂;
    fex ← verify_fcs feu sx;
    SOME (sx,fex)
  od
```

Figure 4.1: Nominal unification.

Lemma 33. *A term is equivalent under different permutations if names in the disagreement don't appear free (2.8)*

$\vdash (\forall a.\ a \in \texttt{dis_set}\ \pi_1\ \pi_2 \Rightarrow fcs \vdash a \# t) \Rightarrow fcs \vdash \pi_1 \bullet t \approx \pi_2 \bullet t$

4.4.1 Soundness

Our basic strategy is to prove Theorem 6, at the end of this section, by recursion induction on $\texttt{unify}_\texttt{n}$. We begin with lemmas that are used in the more difficult cases of that induction.

The first lemma is for the case of unifying suspensions of equal variables, and we show that the helper function `unify_eq_vars` produces a suitable freshness environment when it succeeds.

Lemma 34. *The freshness environment computed by* `unify_eq_vars` *makes the suspensions equivalent*

$$\vdash \texttt{wfs}_\texttt{n}\ s \wedge \texttt{unify_eq_vars}\ (\texttt{dis_set}\ \pi_1\ \pi_2)\ v\ (s,fe) = \texttt{SOME}\ (s,fe') \Rightarrow \\ fe' \vdash s \lhd_\texttt{n} \texttt{Sus}\ \pi_1\ v \approx s \lhd_\texttt{n} \texttt{Sus}\ \pi_2\ v$$

Proof. By case analysis on $\texttt{vwalk}_\texttt{n}\ s\ \texttt{[]}\ v$ (using Lemma 26). In each case we use Lemma 33 and the following lemma which can be proved by finite set induction on ds:

$$\vdash \texttt{FINITE}\ ds \wedge \texttt{unify_eq_vars}\ ds\ v\ (s,fe) = \texttt{SOME}\ (s',fe') \wedge a \in ds \Rightarrow \\ fe' \vdash a \# s' \lhd_\texttt{n} \texttt{Sus}\ \texttt{[]}\ v$$

□

The following lemma is used in the case of unifying two ties with different bound names.

Lemma 35. `verify_fcs` *extends freshness to terms under the new substitution*

$$\vdash \texttt{wfs}_\texttt{n}\ s \wedge fe \vdash a \# t \wedge \texttt{FINITE}\ fe \wedge \texttt{verify_fcs}\ fe\ s = \texttt{SOME}\ fex \Rightarrow \\ fex \vdash a \# s \lhd_\texttt{n} t$$

Proof. By nominal term induction, using Lemma 32. □

The next two lemmas are used in the case of unifying two pairs. The first one is similar to the one above.

Lemma 36. `verify_fcs` *extends equivalence to terms under the new substitution*

$$\vdash fe \vdash t_1 \approx t_2 \wedge \texttt{wfs}_\texttt{n}\ s \wedge \texttt{FINITE}\ fe \wedge \texttt{verify_fcs}\ fe\ s = \texttt{SOME}\ fex \Rightarrow \\ fex \vdash s \lhd_\texttt{n} t_1 \approx s \lhd_\texttt{n} t_2$$

Proof. By rule induction on `equiv`, using Lemma 35. □

The second one, Lemma 37, resolves the following difficulty in the induction for Theorem 6. We need to show that the final unifier makes the pairs equivalent. For the first halves of the pairs, however, the inductive hypotheses only say they are equivalent under an intermediate unifier computed for the first halves only. The substitution of the final unifier is an extension of the intermediate substitution, but the freshness environment is not simply a superset of the intermediate freshness environment, because it has been through two `verify_fcs` stages. Thus we need Lemma 37 to help us connect the final freshness environment to the intermediate one.

Lemma 37. *The result of* `verify_fcs` *in a submap can be verified in the extension*

$$\begin{array}{l}\vdash \texttt{verify_fcs}\ fe\ s = \texttt{SOME}\ ve_0 \land \texttt{verify_fcs}\ fe\ sx = \texttt{SOME}\ ve \land \\ \quad s \sqsubseteq sx \land \texttt{wfs}_{\texttt{n}}\ sx \land \texttt{FINITE}\ fe \Rightarrow \\ \qquad \texttt{verify_fcs}\ ve_0\ sx = \texttt{SOME}\ ve\end{array}$$

Proof. By finite set induction on *fe*. □

Corollary 4. `verify_fcs` *with a fixed substitution is idempotent*

We can now prove the soundness theorem.

Theorem 6. *The result of* `nomunify` *is a unifier, the freshness environment is finite, and the substitution is a well-formed extension*

$$\begin{array}{l}\vdash \texttt{wfs}_{\texttt{n}}\ s \land \texttt{FINITE}\ fe \land \texttt{nomunify}\ (s,fe)\ t_1\ t_2 = \texttt{SOME}\ (sx,fex) \Rightarrow \\ \qquad \texttt{FINITE}\ fex \land \texttt{wfs}_{\texttt{n}}\ sx \land s \sqsubseteq sx \land fex \vdash sx \lhd_{\texttt{n}} t_1 \approx sx \lhd_{\texttt{n}} t_2\end{array}$$

Proof. By recursion induction on $\texttt{unify}_{\texttt{n}}$ using the lemmas above. □

4.4.2 Generality

The statement of generality for the substitution part of the unifier is similar to that of the first-order case. We only consider "in context" generality here, since it generalises the "empty context" notion. Generality of the freshness environment in context means the freshness constraints returned by `nomunify` either came from the initial freshness environment, or would be required in any freshness environment that equates the terms. We define a function `equiv_fcs` that mirrors the definition of `equiv` and determines all the freshness constraints that would be necessary to equate two terms (returning `NONE` if the terms cannot be equivalent). The following lemma characterises `equiv_fcs`.

Lemma 38. `equiv_fcs` *calculates a minimal freshness environment required to equate two terms*

$$\vdash fe \vdash t_1 \approx t_2 \Rightarrow \exists fe_0.\ \texttt{equiv_fcs}\ t_1\ t_2 = \texttt{SOME}\ fe_0 \land fe_0 \subseteq fe$$

We can now use `equiv_fcs` in the statement of generality for the freshness environment part of the result of `nomunify`. The first disjunct in the conclusion of Lemma 39 says that the constraint (a,v) was generated from the constraint (b,w) in the input freshness environment. More specifically, (a,v) is required to ensure that b is fresh for whatever w is bound to in the output substitution sx (we can use `term_fcs` to express this thanks to Lemmas 29–31).

Lemma 39. *A freshness constraint generated by* `nomunify` *is either from the input environment or is required to equate the terms*

$$\vdash \texttt{nomunify}\ (s, fe)\ t_1\ t_2 = \texttt{SOME}\ (sx, fex) \land (a, v) \in fex \land \texttt{wfs}_\texttt{n}\ s \land \texttt{FINITE}\ fe \Rightarrow (\exists b\ w\ fcs.\ (b, w) \in fe \land \texttt{term_fcs}\ b\ (sx \lhd_\texttt{n} \texttt{Sus []}\ w) = \texttt{SOME}\ fcs \land (a, v) \in fcs) \lor (a, v) \in \texttt{THE}\ (\texttt{equiv_fcs}\ (sx \lhd_\texttt{n} t_1)\ (sx \lhd_\texttt{n} t_2))$$

Proof. By recursion induction on $\texttt{unify}_\texttt{n}$. □

Finally, we have the full generality theorem.

Theorem 7. *The result of* `nomunify` *is most general*

$$\vdash \texttt{wfs}_\texttt{n}\ s \land \texttt{FINITE}\ fe \land \texttt{nomunify}\ (s, fe)\ t_1\ t_2 = \texttt{SOME}\ (sx, fex) \land \texttt{wfs}_\texttt{n}\ s_2 \land fe_2 \vdash s_2 \lhd_\texttt{n} s \lhd_\texttt{n} t_1 \approx s_2 \lhd_\texttt{n} s \lhd_\texttt{n} t_2 \Rightarrow (\forall a\ v.\ (a, v) \in fex \Rightarrow (\exists b\ w\ fcs.\ (b, w) \in fe \land \texttt{term_fcs}\ b\ (sx \lhd_\texttt{n} \texttt{Sus []}\ w) = \texttt{SOME}\ fcs \land (a, v) \in fcs) \lor fe_2 \vdash a \# s_2 \lhd_\texttt{n} \texttt{Sus []}\ v) \land \forall t.\ fe_2 \vdash s_2 \lhd_\texttt{n} sx \lhd_\texttt{n} t \approx s_2 \lhd_\texttt{n} s \lhd_\texttt{n} t$$

Proof. The second disjunct of the conclusion is analogous to Lemma 24, and the proof is similar.

The first disjunct is proved using Lemma 39. We use the second disjunct of this theorem and Lemma 38 to show that (s_2, fe_2) must satisfy the constraints in *fex* that didn't come from the input environment. □

4.4.3 Completeness

Lemma 40. *The freshness environment generated by one side of a pair will verify in the substitution computed for both sides.*

$$\vdash fe \vdash t_1 \approx t_2 \land \texttt{term_fcs}\ a\ t_1 = \texttt{SOME}\ fcs_1 \land fcs_1 \subseteq fe \Rightarrow \exists fcs_2.\ \texttt{term_fcs}\ a\ t_2 = \texttt{SOME}\ fcs_2 \land fcs_2 \subseteq fe$$

Proof. By rule induction on `equiv`. □

Theorem 8. *If the terms are unifiable, then* `nomunify` *succeeds*

$$\vdash fe_2 \vdash s_2 \lhd_{\mathtt{n}} s \lhd_{\mathtt{n}} t_1 \approx s_2 \lhd_{\mathtt{n}} s \lhd_{\mathtt{n}} t_2 \wedge \mathtt{wfs_n}\ s_2 \wedge \mathtt{wfs_n}\ s \Rightarrow$$
$$\exists\, sx.$$
$$\forall\, fe.$$
$$\mathtt{FINITE}\ fe \wedge \mathtt{verify_fcs}\ fe\ sx \neq \mathtt{NONE} \Rightarrow$$
$$\exists\, fex.\ \mathtt{nomunify}\ (s, fe)\ t_1\ t_2 = \mathtt{SOME}\ (sx, fex)$$

We can always provide the empty set for the input freshness environment, since $\mathtt{verify_fcs}\ \{\}\ s = \mathtt{SOME}\ \{\}$.

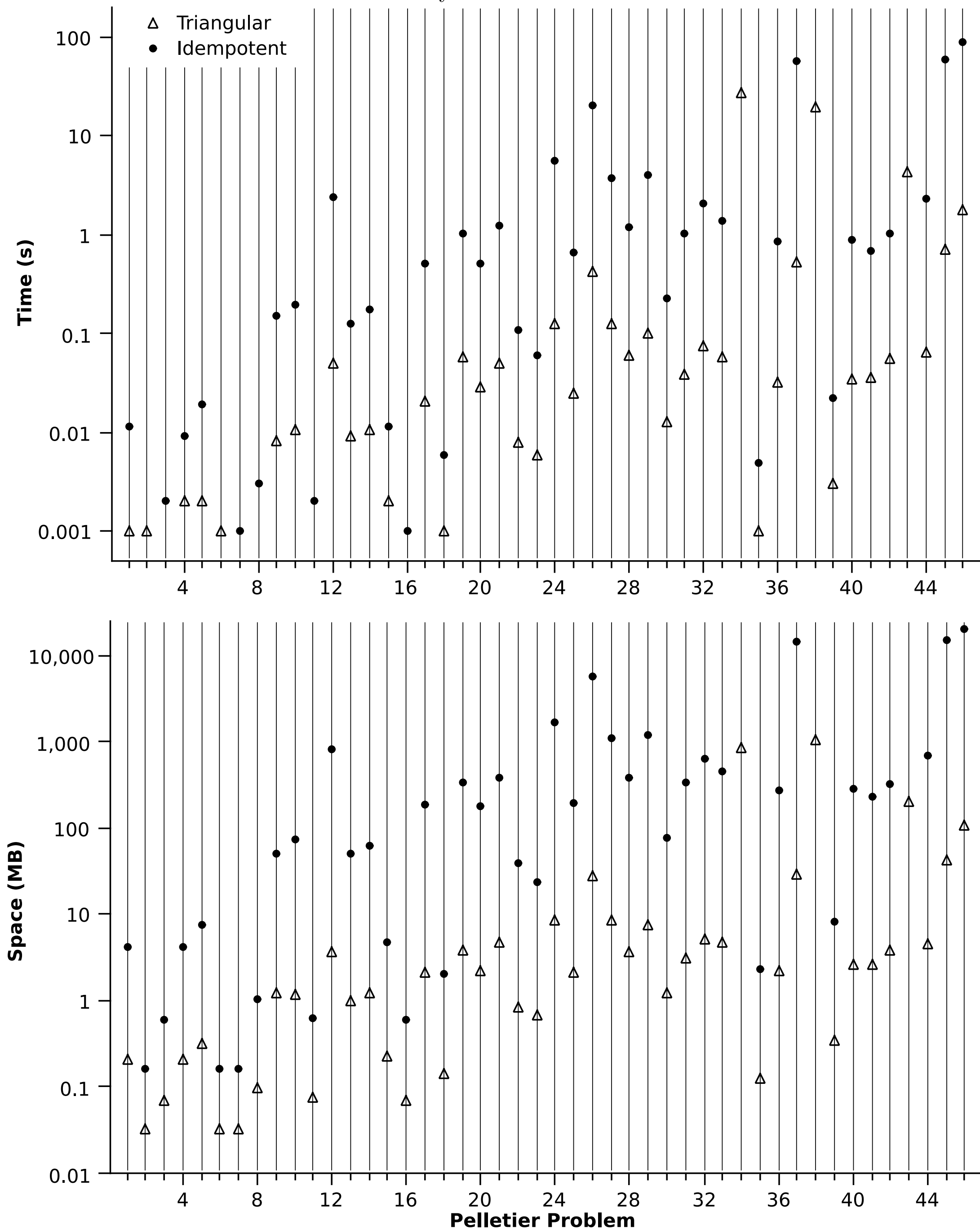
Proof. By recursion induction on $\mathtt{unify_n}$; the proof is similar to that of Theorem 5. Since `unify` extends but otherwise ignores the input freshness environment, we assume the input freshness environment is empty for the inductive proof. We use Lemma 40 in the recursive case. □

4.5 Benchmarks

The nominal unification algorithm from Section 4.3 was implemented in a version of αKanren, which was used in [Near et al. 2008] to write a declarative tableaux theorem prover, αleanTAP. The theorem prover can also be written in αKanren as described in [Byrd and Friedman 2007], which uses idempotent substitutions, because switching to idempotent substitutions doesn't change the language primitives on which the prover depends. Therefore we can compare the triangular and idempotent versions of αKanren empirically by running αleanTAP atop αKanren on a selection of problems for automated theorem provers [Pelletier 1986]. We can see the results in Figure 4.2 on page 59.

The top graph shows the average CPU time (i.e. system plus user time) for each Pelletier problem of the triangular and idempotent substitution versions of αleanTAP. The bottom graph shows the average memory allocation. The averages were calculated from 3 samples on a machine with two Intel Pentium 4 CPUs at 3.8 GHz each and 1 GB of RAM running Ikarus Scheme 0.0.4-rc1+ revision 1747. Problems 3, 7, 8, 11, and 16 did not take the triangular version a measurable time to complete, and problems 2 and 6 likewise for the idempotent version. Problems 34, 38, and 43 were not completed by the idempotent version in under 2 minutes, partly because of thrashing. Both graphs use a logarithmic scale. There is an improvement in the triangular version in all problems except 2 and 6. On average the triangular version is 25 times as fast and allocates 91 times less memory as the idempotent version.

Figure 4.2: Performance of αleanTAP built on αKanren using triangular versus idempotent substitutions, as measured on 46 Pelletier problems. On average the triangular version is 25 times faster and uses 91 times less memory.



Chapter 5

miniKanren Semantics

5.1 Introduction

The source code of a miniKanren program is a structured piece of text that can be interpreted as a theory (plus a query) in first-order logic, and can also be executed by the miniKanren system resulting in a list of answers (or divergence). Conceptually, we can separate the text itself from its meaning either in logic or as instructions for the system. A *semantics* is a rule for giving such a piece of text a meaning; sometimes the meaning itself is also called semantics. There are many kinds of semantics, and approaches to defining them. The interpretation of miniKanren programs as formulas, for example turning

```
(run* (q) (conde ((≡ q 3)) ((≡ q 4) (≡ 3 q))))
```

into

$$\exists q.\, q = 3 \lor (q = 4 \land 3 = q)$$

could be seen as a semantics, although it is not what we usually mean because it is "purely syntactic": the translation is very simple. A better example is the mapping from $\lor$ above to the "or" concept, and from $=$ to "equality". We will use this kind of mapping when we give miniKanren programs a high-level logical semantics.

The point of semantics is to be precise about what our formulas (or programs) mean, because doing so enables consideration of the correctness and other properties of systems that work on the formulas. For example, if we were to write a compiler for miniKanren, we could use a semantics to assess its correctness, and to say which program transformations (such as optimisations) don't change the meaning of a program. The main issues to consider after specifying at least two semantics, one defining correct answers and another giving a system for deriving answers from formulas, are:

- Soundness: Are all the answers derivable in the system correct?
- Completeness: Can all the correct answers be derived in the system?
- Fairness (for a deterministic system): If a derivation exists, will the answer necessarily be found?

We only prove a soundness theorem in this chapter, leaving the other questions to future work.

We will look at three levels of semantics for miniKanren.

1. A high-level logical semantics that specifies when (the formula corresponding to) a program is satisfiable. This is the semantics that defines the set of correct answers to a program. We consider formulas with a single free variable, the query variable, and define answer terms as those terms to which the query variable can be bound that make the formula satisfiable.

2. An operational semantics that specifies an abstract machine that works on a program and may generate answers. The meaning of a program is all the runs of the machine that start with the initial machine state corresponding to the program. We will consider a possibly non-deterministic machine, so there may be many possible runs. We will, however, allow the machine to be made deterministic in different ways, so that we might use this semantics to investigate different evaluation strategies.

3. An executable semantics, translating programs to ML code that closely matches the Scheme implementation of miniKanren. Connecting this semantics to the first one, possibly via the second, would be a way of showing that interleaving streams search strategy used in the implementation is correct.

5.2 Syntax

Any talk of semantics depends first on fixing the kinds of text to which meaning is going to be ascribed. For programming languages and other formal systems, this is called the syntax. In this section we describe our model of miniKanren syntax in HOL. We use inductive data types to model syntactic constructs, because they are natural fit to the tree-structured text of our programs.

The only values (data types) in miniKanren are terms. We call the syntax intended for constructing a term an *expression* (Definition 33), since in Scheme terminology it is the evaluation of expressions that yields values. The expression type is essentially the same as the term type in Definition 1, except the variables are strings. We use strings here because variables in expressions denote references to lexical variables in a program, and they are referred to by name. We must distinguish between lexical variables (`EVar`) and logic variables (`Var`): the latter are hidden from a miniKanren programmer. In Listing 1.5 (page 6), for example, l s and ls are all lexical variables that may or may not be bound to logic variables, depending on the arguments with which `appendo` was called.

Definition 33. Expressions

```
α expr
  = EVar of string | EPair of α expr => α expr | EConst of α
```

In our model, a miniKanren program, including definitions, will be represented by a single *goal expression* (Definition 34). Goal expressions are akin to first-order formulas, except they allow scoped definitions of new relations (and, like all the formulas we've seen, don't allow negation, implication, or universal quantification).

Definition 34. Goal expressions (formulas)

```
α gexpr
  = Conj of α gexpr => α gexpr
  | Disj of α gexpr => α gexpr
  | LetVar of string => α gexpr
  | LetVal1 of string # α expr => α gexpr
  | Let1 of string # string list # α gexpr => α gexpr
  | LetRec1 of (string, string list # α gexpr) alist => α gexpr
  | Call of string => α expr list
```

The goal expression constructors are most easily explained by an example. Listing 5.1 shows a miniKanren program (using **letrec** instead of **define** so that the whole program is in one form) that includes examples of all the syntactic constructs. The corresponding goal expression is given in Figure 5.1 on page 64. (The answer list computed for this program is (⟨5, ()⟩)). A version of the syntax that uses a `Conde` constructor directly, instead of `Conj` and `Disj`, and `Exist` instead of nested `LetVar`s, could be defined as syntactic sugar over goal expressions.

A miniKanren program is represented by a goal expression together with a query variable that may appear free in the goal. Any other variables must be bound (by `LetVar`). Alternatively, we could say that a program is a `LetVar` expression, with no free variables, where that first binding is taken as the query variable.

Well-formed programs We restrict our attention to *well-formed* programs that don't contain free variables and whose `Call` expressions always supply the right number of arguments. The functions `wf_expr` and `wf_gexpr` check for well-formedness with respect to a set *bvs* of bound variables, and, for the latter, a finite map *brs* from relation names to their arities. The last line of `wf_gexpr`'s definition shows that there is one relation, `"Eqv"` (*i.e.*, ≡), that is always bound, and has arity 2 if it hasn't been bound to something else by *brs*. The function `(o_a)` is composition of a function over an association-list: since the elements of *defs* are tuples `(`*name*`,`*fmls*`,`*body*`)`, the term `LENGTH ∘ FST o_a` *defs* is a list of elements `(`*name*`,LENGTH` *fmls*`)`.

Definition 35. Well-formed expressions

$$\texttt{wf_expr}\ bvs\ (\texttt{EVar}\ v) \iff v \in bvs$$
$$\texttt{wf_expr}\ bvs\ (\texttt{EPair}\ e_1\ e_2) \iff \texttt{wf_expr}\ bvs\ e_1 \land \texttt{wf_expr}\ bvs\ e_2$$
$$\texttt{wf_expr}\ bvs\ (\texttt{EConst}\ c) \iff \texttt{T}$$

```
(let ((n ())
      (fiveo (λ (x) (≡ x 5))))
  (letrec ((fivelisto
             (λ (ls) (conde ((≡ ls ()))
                            ((exist (a d)
                               (≡ ls ⟨a, d⟩)
                               (fiveo a)
                               (fivelisto d)))))))
    (run* (q) (exist (x)
                (fivelisto q)
                (≡ q ⟨x, n⟩)))))
```

Listing 5.1: A miniKanren program illustrating all syntactic constructs.

```
LetVal1 ("n",EConst "()")
  (Let1 ("fiveo",["x"],Call "Eqv" [EVar "x"; EConst "5"])
     (LetRec1
        [("fivelisto",["ls"],
          Disj (Call "Eqv" [EVar "ls"; EConst "()"])
             (LetVar "a"
                (LetVar "d"
                   (Conj
                      (Call "Eqv"
                         [EVar "ls"; EPair (EVar "a") (EVar "d")])
                      (Conj (Call "fiveo" [EVar "a"])
                         (Call "fivelisto" [EVar "d"]))))))]
        (LetVar "x"
           (Conj (Call "fivelisto" [EVar "q"])
              (Call "Eqv"
                 [EVar "q"; EPair (EVar "x") (EVar "n")])))))
```

Figure 5.1: Goal expression corresponding to Listing 5.1.

Definition 36. Well-formed formulas

```
wf_gexpr brs bvs (Conj g1 g2) ⟺
  wf_gexpr brs bvs g1 ∧ wf_gexpr brs bvs g2
wf_gexpr brs bvs (Disj g1 g2) ⟺
  wf_gexpr brs bvs g1 ∧ wf_gexpr brs bvs g2
wf_gexpr brs bvs (LetVar v g) ⟺ wf_gexpr brs (v INSERT bvs) g
wf_gexpr brs bvs (LetVal1 (v,x) g) ⟺
  wf_expr bvs x ∧ wf_gexpr brs (v INSERT bvs) g
wf_gexpr brs bvs (Let1 (name,fmls,body) g) ⟺
  wf_gexpr (brs |+ (name,LENGTH fmls)) bvs g ∧ ALL_DISTINCT fmls ∧
  wf_gexpr brs (bvs ∪ set fmls) body
wf_gexpr brs bvs (LetRec1 defs g) ⟺
  ALL_DISTINCT (MAP FST defs) ∧
  (let ebrs = brs |++ (LENGTH ∘ FST o_a defs) in
     wf_gexpr ebrs bvs g ∧
     EVERY
       (λ (name,fmls,body).
          ALL_DISTINCT fmls ∧ wf_gexpr ebrs (bvs ∪ set fmls) body)
       defs)
wf_gexpr brs bvs (Call name args) ⟺
  EVERY (wf_expr bvs) args ∧
  (brs ' name = SOME (LENGTH args) ∨
   name = "Eqv" ∧ name ∉ FDOM brs ∧ LENGTH args = 2)
```

5.3 Logical semantics

The meaning of a goal expression is with respect to two *environments*. One environment provides bindings for the free variables, and the other provides bindings for the relations that might be called. We use the first kind of environment in the evaluation of expressions to terms (Definition 37). The HOL term *env* ' *v* in the first clause simply means THE (*env* ' *v*).

Definition 37. Evaluating an expression

```
evalex env (EVar v) = env ' v
evalex env (EPair e1 e2) = ⟨evalex env e1,evalex env e2⟩
evalex env (EConst c) = Const c
```

The second kind of environment, which we call a definition environment, binds the names of relations to *closures*. A closure consists of the goal expression making up the body of the relation, the names of the formal parameters (arguments), and two environments to be used whenever the relation is called. The environments in the closure provide bindings for any free variables or relations that appear in the body.

The mapping from relation names to closures is encoded across two types, α `denv` and α `defs`, and the lookup function `DLOOKUP` (Definition 40). The lookup function

takes a definition environment and a name to look up, and returns `NONE` if the name isn't bound, otherwise `SOME (`*denv*`,`*env*`,`*defs*`,`*fmls*`,`*body*`)`. The first two components of the result are the two environments of the closure, and the last two components are the formal parameter list and the body goal expression. The middle component, *defs*, contains signatures of all the relations that were defined at the same time as the one that was looked up. For a non-recursive relation, this will just be the relation itself. But for a recursive relation, there may be other mutually recursive relations defined at the same time, and we will use the *defs* part of the result to keep them in the scope of *body*. We distinguish between recursive and non-recursive relations, to allow evaluation strategies to treat them differently if necessary, for example by jumping to a different branch of an interleaving search tree on calls to recursive relations only.

Definition 38. Definitions

```
α defs
  = Rec of string ↦ string list # α gexpr
  | NonRec of string # string list # α gexpr
```

Definition 39. Definition environments

```
α denv = DEMPTY | DUPDATE of α denv => α env => α defs
```

Definition 40. Looking up a relation

```
DLOOKUP DEMPTY name = NONE
DLOOKUP (DUPDATE denv env defs) name =
  case
    case defs of
      Rec dict → dict ' name
    || NonRec (xname,fmls,body) →
        if name = xname then SOME (fmls,body) else NONE
    of
    NONE → DLOOKUP denv name
  || SOME (fmls,body) → SOME (denv,env,defs,fmls,body)
```

We can now define a relation, `sat`, that specifies whether or not a goal expression is satisfiable (Definition 41).

Definition 41. Satisfying a formula

$$\frac{\texttt{sat}\ \mathit{denv}\ \mathit{env}\ g_1 \quad \texttt{sat}\ \mathit{denv}\ \mathit{env}\ g_2}{\texttt{sat}\ \mathit{denv}\ \mathit{env}\ (\texttt{Conj}\ g_1\ g_2)}$$

$$\frac{\texttt{sat}\ \mathit{denv}\ \mathit{env}\ g_1}{\texttt{sat}\ \mathit{denv}\ \mathit{env}\ (\texttt{Disj}\ g_1\ g_2)} \qquad \frac{\texttt{sat}\ \mathit{denv}\ \mathit{env}\ g_2}{\texttt{sat}\ \mathit{denv}\ \mathit{env}\ (\texttt{Disj}\ g_1\ g_2)}$$

$$\frac{\texttt{sat}\ \mathit{denv}\ (\mathit{env}\ \texttt{|+}\ (v\texttt{,}t))\ g}{\texttt{sat}\ \mathit{denv}\ \mathit{env}\ (\texttt{LetVar}\ v\ g)} \qquad \frac{\texttt{sat}\ \mathit{denv}\ (\mathit{env}\ \texttt{|+}\ (v\texttt{,evalex}\ \mathit{env}\ x))\ g}{\texttt{sat}\ \mathit{denv}\ \mathit{env}\ (\texttt{LetVal1}\ (v\texttt{,}x)\ g)}$$

$$\frac{\texttt{sat}\ (\texttt{DUPDATE}\ \mathit{denv}\ \mathit{env}\ (\texttt{NonRec}\ (\mathit{name}\texttt{,}\mathit{fmls}\texttt{,}\mathit{body})))\ \mathit{env}\ g}{\texttt{sat}\ \mathit{denv}\ \mathit{env}\ (\texttt{Let1}\ (\mathit{name}\texttt{,}\mathit{fmls}\texttt{,}\mathit{body})\ g)}$$

$$\frac{\texttt{sat}\ (\texttt{DUPDATE}\ \mathit{denv}\ \mathit{env}\ (\texttt{Rec}\ (\texttt{FEMPTY}\ \texttt{|++}\ \mathit{defs})))\ \mathit{env}\ g}{\texttt{sat}\ \mathit{denv}\ \mathit{env}\ (\texttt{LetRec1}\ \mathit{defs}\ g)}$$

$$\frac{\texttt{DLOOKUP}\ \mathit{denv}\ \texttt{"Eqv"} = \texttt{NONE} \quad \texttt{evalex}\ \mathit{env}\ e_1 = \texttt{evalex}\ \mathit{env}\ e_2}{\texttt{sat}\ \mathit{denv}\ \mathit{env}\ (\texttt{Call}\ \texttt{"Eqv"}\ [e_1\texttt{;}\ e_2])}$$

$$\frac{\begin{array}{c}\texttt{DLOOKUP}\ \mathit{denv}\ \mathit{name} = \texttt{SOME}\ (\mathit{cdenv}\texttt{,}\mathit{cenv}\texttt{,Rec}\ d\texttt{,}\mathit{fmls}\texttt{,}\ \mathit{body}) \\ \texttt{sat}\ (\texttt{DUPDATE}\ \mathit{cdenv}\ \mathit{cenv}\ (\texttt{Rec}\ d))\ (\texttt{call_env}\ \mathit{cenv}\ \mathit{fmls}\ (\texttt{evalex}\ \mathit{env})\ \mathit{args})\ \mathit{body}\end{array}}{\texttt{sat}\ \mathit{denv}\ \mathit{env}\ (\texttt{Call}\ \mathit{name}\ \mathit{args})}$$

$$\frac{\begin{array}{c}\texttt{DLOOKUP}\ \mathit{denv}\ \mathit{name} = \texttt{SOME}\ (\mathit{cdenv}\texttt{,}\mathit{cenv}\texttt{,NonRec}\ d\texttt{,}\mathit{fmls}\texttt{,}\ \mathit{body}) \\ \texttt{sat}\ \mathit{cdenv}\ (\texttt{call_env}\ \mathit{cenv}\ \mathit{fmls}\ (\texttt{evalex}\ \mathit{env})\ \mathit{args})\ \mathit{body}\end{array}}{\texttt{sat}\ \mathit{denv}\ \mathit{env}\ (\texttt{Call}\ \mathit{name}\ \mathit{args})}$$

A `Conj` is `sat` if both sides are `sat`; a `Disj` is `sat` if either side is `sat`. The `LetVal1`, `Let1`, and `LetRec1` goal expressions extend the environments appropriately.

Existential variables, introduced with `LetVar`, are taken to range over terms.[1] A `LetVar` is `sat` if the body is `sat` with the variable bound to some term t; we don't say what t is, and just require that it exists.

Calls to the `"Eqv"` relation are interpreted as equality of the terms resulting from evaluating the arguments. Calls to other relations are interpreted according to the definitions of those relations in the current definition environment. The HOL term `call_env` *cenv fmls eval args* denotes the environment *cenv* extended with bindings for each variable in the list *fmls* to the value obtained by calling *eval* on an expression in the list *args*.

5.4 Operational semantics

The semantics in the previous section gives the logical side of the meaning of an miniKanren program. In this section we develop the computational side. We give an

[1] For those in the know, this means we look for Herbrand interpretations.

operational semantics, which means describing an abstract machine and a transition relation (`step`) from one state of the machine to another. Any miniKanren program will correspond to some initial state of the machine, and the meaning of the program is given by the behaviour of the machine starting from that initial state and following its step relation to other states. We will allow the machine to emit substitutions while it runs, and will examine the possibly infinite stream of substitutions that are emitted.

To describe the states of our machine, we introduce the idea of (dynamic) conjuncts and disjuncts. A conjunct is a goal expression, together with the current environment, that needs to be satisfied alongside any number of other conjuncts. The goal expression of a conjunct will have come from a `Conj` in the original program, but conjuncts from different levels of nesting in the syntax will all be accumulated at the same level in the machine state. A disjunct is a collection of conjuncts paired with the current substitution that they all share. A state of the machine is a collection of disjuncts.

A state is well-formed (`wf_state`) if all its contents are well-formed. Ultimately this means that the variables and relations in the goal expressions are bound in their associated environments, and also that the substitutions associated with each disjunct are well-formed.

Each disjunct can be thought of as a branch in a dynamically growing search tree. The machine moves from one state to another by working on one of the disjuncts (which involves working on one of the conjuncts in that disjunct). Different choices about which disjunct (and which conjunct) to work on next correspond to different search strategies over the tree. To make our machine reusable with different search strategies, we store collections of disjuncts and conjuncts in bags that are parameterised by an element replacement policy.

5.4.1 Structured bags

A structured bag is an abstract container data type that behaves similarly to a finite bag. In particular, no element can be taken out of a structured bag that hasn't first been put in. However, adding and removing elements is constrained by a replacement relation that may only allow certain transformations on the container's contents. We represent a particular kind of structured bag by a *record* in HOL consisting of a distinguished empty bag value, of type γ, and a replacement relation of type $\gamma \to \epsilon\ \texttt{list} \to \epsilon\ \texttt{list} \to \gamma \to \texttt{bool}$ where γ is the type of the container and ϵ the type of the elements. Suppose f is a record representing a kind of structured bag, and c_1 is a container of that kind. Then the relation f`.replace` $c_1\ \ l_1\ \ l_2\ \ c_2$ holds if c_2 can be obtained by removing the elements of l_1 from c_1 and adding the elements of l_2.

The contents of a structured bag can be retrieved with `bag_of_sbag`, which takes the structured bag record and a container and returns an ordinary bag with the same contents as the container. Operations on structured bags that only depend on the bag of elements can be defined in terms of `bag_of_sbag`. For example:

Definition 42. Null test for structured bags

$$\vdash \texttt{sbag_null}\ f\ c \iff \texttt{bag_of_sbag}\ f\ c = \{\!|\,|\!\}$$

There is no HOL type for structured bags; rather we have a predicate `is_sbag` which holds of a record and ensures that the record's replacement relation behaves well (doesn't allow elements to come out that weren't put in). The type of the container that the record's replacement relation operates on can be anything. We use another predicate `sbag_wf` that takes a record and a container and checks whether the container can be obtained by starting from the record's empty container and taking a sequence of element replacements.

5.4.2 The state transition relation

The main component of our operational semantics is the relation that takes us from one machine state to another. We allow the machine to emit answer substitutions at any time while running, and we (fully) apply these answer substitutions to the query variable to obtain the answer terms associated with a run. The first two arguments of the `step` relation (*df* and *cf*) are records specifying the kind of structured bags containing the disjuncts and conjuncts in the state, respectively. The third and fifth arguments are machine states: the state before (*state*) and after (*cont*) taking a step. The fourth argument (*ans*) is an optional emitted substitution. The `step` relation is defined (as an inductive relation) in Definition 43 (Figures 5.4.2 and 5.4.2).

The machine emits the substitution s associated with a disjunct `(`s`,`cs`)` when cs is empty, and discards the disjunct (since it can produce no more answer substitutions). The first clause of Definition 43 describes this behaviour, and is the only clause that emits a substitution. The conjuncts in a disjunct represent goals to be solved in order to satisfy the disjunct. Thus it makes sense to emit the substitution that was built up as the goals were solved when no conjuncts are left.

The remaining clauses all work on a disjunct that has conjuncts remaining. In each clause, we replace a disjunct `(`s`,`cs`)` with a replacement obtained by replacing a conjunct in cs. The replacements depend on the type of goal expression found in the conjunct to be replaced. For example, we replace a conjunct `(`d`,`e`,Conj` g_1 g_2`)` by two conjuncts `(`d`,`e`,`g_1`)` and `(`d`,`e`,`g_2`)`, meaning that both halves of the `Conj` must be solved. In the `LetVar` case we add a binding for the lexical variable to a new logic variable `Var` n. We don't specify the number n, but require that it doesn't appear anywhere else in the disjunct. Thus underspecification is a source of non-determinism in the machine, in addition to the non-determinism in picking which conjunct of which disjunct to work on. In the case of a call to `"Eqv"`, when `"Eqv"` hasn't been rebound, we call `unify` using the disjunct's substitution as initial substitution. If unification succeeds we remove the `Call` conjunct and update the disjunct's substitution; if unification fails we discard the disjunct.

Definition 43. One step transition relation

$$\frac{\texttt{sbag_null}\ \mathit{cf}\ \ \mathit{cs}\quad \mathit{df}\texttt{.replace}\ \mathit{state}\ \texttt{[(}s\texttt{,}\mathit{cs}\texttt{)]}\ \texttt{[]}\ \mathit{cont}}{\texttt{step}\ \mathit{df}\ \ \mathit{cf}\ \ \mathit{state}\ \texttt{(SOME}\ s\texttt{)}\ \mathit{cont}}$$

$$\frac{\begin{array}{c}\mathit{cf}\texttt{.replace}\ \mathit{cs}\ \texttt{[(}d\texttt{,}e\texttt{,Conj}\ g_1\ \ g_2\texttt{)]}\ \texttt{[(}d\texttt{,}e\texttt{,}g_1\texttt{);}\ \texttt{(}d\texttt{,}e\texttt{,}g_2\texttt{)]}\ \mathit{ccs}\\ \mathit{df}\texttt{.replace}\ \mathit{state}\ \texttt{[(}s\texttt{,}\mathit{cs}\texttt{)]}\ \texttt{[(}s\texttt{,}\mathit{ccs}\texttt{)]}\ \mathit{cont}\end{array}}{\texttt{step}\ \mathit{df}\ \ \mathit{cf}\ \ \mathit{state}\ \texttt{NONE}\ \mathit{cont}}$$

$$\frac{\begin{array}{c}\mathit{cf}\texttt{.replace}\ \mathit{cs}\ \texttt{[(}d\texttt{,}e\texttt{,Disj}\ g_1\ \ g_2\texttt{)]}\ \texttt{[(}d\texttt{,}e\texttt{,}g_1\texttt{)]}\ \mathit{ccs}_1\\ \mathit{cf}\texttt{.replace}\ \mathit{cs}\ \texttt{[(}d\texttt{,}e\texttt{,Disj}\ g_1\ \ g_2\texttt{)]}\ \texttt{[(}d\texttt{,}e\texttt{,}g_2\texttt{)]}\ \mathit{ccs}_2\\ \mathit{df}\texttt{.replace}\ \mathit{state}\ \texttt{[(}s\texttt{,}\mathit{cs}\texttt{)]}\ \texttt{[(}s\texttt{,}\mathit{ccs}_1\texttt{);}\ \texttt{(}s\texttt{,}\mathit{ccs}_2\texttt{)]}\ \mathit{cont}\end{array}}{\texttt{step}\ \mathit{df}\ \ \mathit{cf}\ \ \mathit{state}\ \texttt{NONE}\ \mathit{cont}}$$

$$\frac{\begin{array}{c}n \notin \texttt{disjunct_vars}\ \mathit{cf}\ \ \texttt{(}s\texttt{,}\ \mathit{cs}\texttt{)}\\ \mathit{cf}\texttt{.replace}\ \mathit{cs}\ \texttt{[(}d\texttt{,}e\texttt{,LetVar}\ v\ \ g\texttt{)]}\ \texttt{[(}d\texttt{,}e\ \texttt{|+}\ \texttt{(}v\texttt{,Var}\ n\texttt{),}g\texttt{)]}\ \mathit{ccs}\\ \mathit{df}\texttt{.replace}\ \mathit{state}\ \texttt{[(}s\texttt{,}\mathit{cs}\texttt{)]}\ \texttt{[(}s\texttt{,}\mathit{ccs}\texttt{)]}\ \mathit{cont}\end{array}}{\texttt{step}\ \mathit{df}\ \ \mathit{cf}\ \ \mathit{state}\ \texttt{NONE}\ \mathit{cont}}$$

$$\frac{\begin{array}{c}\mathit{cf}\texttt{.replace}\ \mathit{cs}\ \texttt{[(}d\texttt{,}e\texttt{,LetVal1}\ \texttt{(}v\texttt{,}x\texttt{)}\ \ g\texttt{)]}\ \texttt{[(}d\texttt{,}e\ \texttt{|+}\ \texttt{(}v\texttt{,evalex}\ \ e\ \ x\texttt{),}g\texttt{)]}\ \mathit{ccs}\\ \mathit{df}\texttt{.replace}\ \mathit{state}\ \texttt{[(}s\texttt{,}\mathit{cs}\texttt{)]}\ \texttt{[(}s\texttt{,}\mathit{ccs}\texttt{)]}\ \mathit{cont}\end{array}}{\texttt{step}\ \mathit{df}\ \ \mathit{cf}\ \ \mathit{state}\ \texttt{NONE}\ \mathit{cont}}$$

$$\frac{\begin{array}{c}\mathit{cf}\texttt{.replace}\ \mathit{cs}\ \texttt{[(}d\texttt{,}e\texttt{,Let1}\ \mathit{def}\ \ g\texttt{)]}\ \texttt{[(DUPDATE}\ d\ \ e\ \texttt{(NonRec}\ \mathit{def}\texttt{),}e\texttt{,}g\texttt{)]}\ \mathit{ccs}\\ \mathit{df}\texttt{.replace}\ \mathit{state}\ \texttt{[(}s\texttt{,}\mathit{cs}\texttt{)]}\ \texttt{[(}s\texttt{,}\mathit{ccs}\texttt{)]}\ \mathit{cont}\end{array}}{\texttt{step}\ \mathit{df}\ \ \mathit{cf}\ \ \mathit{state}\ \texttt{NONE}\ \mathit{cont}}$$

$$\frac{\begin{array}{c}\mathit{cf}\texttt{.replace}\ \mathit{cs}\ \texttt{[(}d\texttt{,}e\texttt{,LetRec1}\ \mathit{ls}\ \ g\texttt{)]}\ \texttt{[(DUPDATE}\ d\ \ e\ \texttt{(Rec (FEMPTY |++}\ \mathit{ls}\texttt{)),}e\texttt{,}g\texttt{)]}\ \mathit{ccs}\\ \mathit{df}\texttt{.replace}\ \mathit{state}\ \texttt{[(}s\texttt{,}\mathit{cs}\texttt{)]}\ \texttt{[(}s\texttt{,}\mathit{ccs}\texttt{)]}\ \mathit{cont}\end{array}}{\texttt{step}\ \mathit{df}\ \ \mathit{cf}\ \ \mathit{state}\ \texttt{NONE}\ \mathit{cont}}$$

Figure 5.2: Transition relation for the abstract machine defined as an operational semantics for miniKanren. The first clause emits the substitution associated with a disjunct with no remaining conjuncts. The other clauses, one for each goal expression constructor specify a transformation to be made on some conjunct associated with some disjunct in the current machine state. The clauses for `Call`, including one each for successful and unsuccessful calls to `"Eqv"`, are in Figure 5.4.2 on page 71.

$$\frac{\begin{array}{c} cf.\texttt{replace}\ cs\ \texttt{[(}d\texttt{,}e\texttt{,Call "Eqv" [}e_1\texttt{; }e_2\texttt{])] []}\ ccs \\ \texttt{DLOOKUP}\ d\ \texttt{"Eqv"} = \texttt{NONE} \\ \texttt{unify}\ s\ \texttt{(evalex}\ e\ e_1\texttt{)}\ \texttt{(evalex}\ e\ e_2\texttt{)} = \texttt{SOME}\ sx \\ df.\texttt{replace}\ state\ \texttt{[(}s\texttt{,}cs\texttt{)] [(}sx\texttt{,}ccs\texttt{)]}\ cont \end{array}}{\texttt{step}\ df\ cf\ state\ \texttt{NONE}\ cont}$$

$$\frac{\begin{array}{c} cf.\texttt{replace}\ cs\ \texttt{[(}d\texttt{,}e\texttt{,Call "Eqv" [}e_1\texttt{; }e_2\texttt{])] []}\ ccs \\ \texttt{DLOOKUP}\ d\ \texttt{"Eqv"} = \texttt{NONE} \\ \texttt{unify}\ s\ \texttt{(evalex}\ e\ e_1\texttt{)}\ \texttt{(evalex}\ e\ e_2\texttt{)} = \texttt{NONE} \\ df.\texttt{replace}\ state\ \texttt{[(}s\texttt{,}cs\texttt{)] []}\ cont \end{array}}{\texttt{step}\ df\ cf\ state\ \texttt{NONE}\ cont}$$

$$\frac{\begin{array}{c} cf.\texttt{replace}\ cs\ \texttt{[(}d\texttt{,}e\texttt{,Call}\ name\ args\texttt{)] [(DUPDATE}\ cd\ ce\ \texttt{(Rec}\ fm\texttt{),call_env}\ ce\ f\ \texttt{(evalex}\ e\texttt{)}\ args\texttt{,}g\texttt{)]}\ ccs \\ \texttt{DLOOKUP}\ d\ name = \texttt{SOME (}cd\texttt{,}ce\texttt{,Rec}\ fm\texttt{,}f\texttt{, }g\texttt{)} \\ df.\texttt{replace}\ state\ \texttt{[(}s\texttt{,}cs\texttt{)] [(}s\texttt{,}ccs\texttt{)]}\ cont \end{array}}{\texttt{step}\ df\ cf\ state\ \texttt{NONE}\ cont}$$

$$\frac{\begin{array}{c} cf.\texttt{replace}\ cs\ \texttt{[(}d\texttt{,}e\texttt{,Call}\ name\ args\texttt{)] [(}cd\texttt{,call_env}\ ce\ f\ \texttt{(evalex}\ e\texttt{)}\ args\texttt{,}g\texttt{)]}\ ccs \\ \texttt{DLOOKUP}\ d\ name = \texttt{SOME (}cd\texttt{,}ce\texttt{,NonRec}\ nfb\texttt{,}f\texttt{, }g\texttt{)} \\ df.\texttt{replace}\ state\ \texttt{[(}s\texttt{,}cs\texttt{)] [(}s\texttt{,}ccs\texttt{)]}\ cont \end{array}}{\texttt{step}\ df\ cf\ state\ \texttt{NONE}\ cont}$$

Figure 5.3: `Call` clauses of Figure 5.4.2 on page 70.

We have defined a single step of the machine; now we look at how to get it started. Given a well-formed goal expression g and a query variable q, representing a miniKanren program, an *initial state* of the machine contains a single disjunct consisting of the empty substitution and a single conjunct consisting of the empty definition environment, the environment with a single binding from q to a variable `Var` n, and the formula g. The logic variable `Var` n corresponds to the lexical variable q; n can be any number.

A run of the machine is a sequence of states beginning at an initial state and moving to new states *via* the `step` relation. Associated with a run is a sequence of emitted substitutions, interspersed with `NONE`s, corresponding to the *ans* argument of each `step`. Runs, and these sequences of optional substitutions, can be infinite, since the machine can get into an infinite loop by calling a relation that generates another call to itself with the same arguments. To model infinite sequences in HOL, we can't use lists, since all lists are finite. Instead, we use lazy lists.

5.4.3 Lazy lists

A lazy list in HOL is either the empty lazy list $[||]$ or an object that can be deconstructed into a head element and a tail lazy list, written $ll = h:::tl$. Lazy lists are the coinductive counterpart to lists; whereas lists are constructed up from empty, lazy lists are deconstructed down towards empty, and this allows them to be infinite. Mechanisation of coinductive types in HOL is described in [Paulson 1997].

We use lazy lists to collect the possibly infinite sequence of *ans* arguments of the `step` relation during a run of our machine. To do this, we define a coinductive relation, `stream_of` (Definition 44), that relates an initial state to a lazy list of answers from an arbitrary `step`-like relation R. (Of course we will usually use `stream_of (step` df cf`)` for some structured bag records df and cf.) The first clause of the definition returns the empty lazy list if no steps can be taken from *state*. The second clause returns $ans:::rest$ if there is a step from *state* to *cont* returning *ans*, and *rest* is the lazy list obtained starting from *cont*.

Definition 44. Possibly infinite streams of steps (a coinductive relation)

$$\frac{\forall\, ans\ cont.\ \neg R\ state\ ans\ cont}{\texttt{stream_of}\ R\ state\ [||]}$$

$$\frac{R\ state\ ans\ cont \quad \texttt{stream_of}\ R\ cont\ rest}{\texttt{stream_of}\ R\ state\ (ans:::rest)}$$

A lazy list can be created by "unfolding" a function of type $\alpha \rightarrow (\alpha\ \#\ \beta)$ `option`. The function is applied to a seed value of type α, and if it returns `NONE`, the list is ended. If the function returns `SOME` (a,b), then the next element of the list is b and the next seed value is a. We continue this process until the function returns `NONE` (which may be never) to generate a lazy list. `LUNFOLD` f a denotes the lazy list obtained by unfolding

f with the seed a in this way.

Lemma 41. *There is at least one lazy list associated with every initial state*

$\vdash \exists\, llist.$ `stream_of` $R\ \ state\ \ llist$

Proof. By `stream_of` coinduction. We construct the lazy list by unfolding a function with initial seed *state*. The function returns `NONE` if there are no more R-steps available, otherwise picks an arbitrary R-step and returns that step's answer and next state. □

There is a coinductive relation `every` in HOL4's lazy list theory, which we will use in our soundness proof. `every` $P\ \ ll$ holds if every element x of the lazy list ll makes $P\ \ x$ true.

5.5 Soundness

The basic outline of the soundness proof is as follows:

- define a notion of satisfiability of machine states by a substitution (Definition 45),
- show that a substitution satisfies the state of the machine immediately before it is emitted (Lemma 42),
- show that a substitution satisfying a state also satisfies the previous state (Lemma 43), and
- show that a substitution satisfying an initial state provides a binding for the query variable that satisfies the corresponding program (Lemma 45).

In summary: we work backwards from the state where a substitution is emitted to show that it must satisfy any program corresponding to the initial state that led to the state of emission.

We begin with the definition of `sat_state` (page 74). A substitution satisfies a machine state if it satisfies at least one of the disjuncts in the state. The disjuncts are in disjunction in the sense that only one of them needs to be satisfied. To satisfy a disjunct, the substitution must satisfy all the conjuncts in the disjunct, and must be an extension of the disjunct's current substitution. The condition that the substitution be an extension of the disjunct's substitution may be surprisingly strong; it is suitable for a soundness proof because we only work back from the substitutions actually emitted and don't need to consider all the substitutions that might satisfy an initial state but aren't computed by the machine. For a substitution to satisfy a conjunct that is a call to `"Eqv"`, the terms obtained by applying the substitution to the arguments of the call must be equal. The other cases of `sat_cjnt` are straightforward.

Definition 45. Satisfaction of a state by a substitution

$\texttt{sat_state}\ \mathit{df}\ \mathit{cf}\ s\ \mathit{state} \iff$
$\quad \exists\, \mathit{dis}.\ \texttt{sbag_mem}\ \mathit{df}\ \mathit{dis}\ \mathit{state} \land \texttt{sat_djnt}\ \mathit{cf}\ s\ \mathit{dis}$

$\texttt{sat_djnt}\ \mathit{cf}\ \mathit{ss}\ (s, \mathit{cs}) \iff$
$\quad \texttt{wfs}\ \mathit{ss} \land s \sqsubseteq \mathit{ss} \land \texttt{sbag_every}\ \mathit{cf}\ (\texttt{sat_cjnt}\ \mathit{ss})\ \mathit{cs}$

$$\frac{\texttt{sat_cjnt}\ s\ (\mathit{denv}, \mathit{env}, g_1) \quad \texttt{sat_cjnt}\ s\ (\mathit{denv}, \mathit{env}, g_2)}{\texttt{sat_cjnt}\ s\ (\mathit{denv}, \mathit{env}, \texttt{Conj}\ g_1\ g_2)}$$

$$\frac{\texttt{sat_cjnt}\ s\ (\mathit{denv}, \mathit{env}, g_1)}{\texttt{sat_cjnt}\ s\ (\mathit{denv}, \mathit{env}, \texttt{Disj}\ g_1\ g_2)} \qquad \frac{\texttt{sat_cjnt}\ s\ (\mathit{denv}, \mathit{env}, g_2)}{\texttt{sat_cjnt}\ s\ (\mathit{denv}, \mathit{env}, \texttt{Disj}\ g_1\ g_2)}$$

$$\frac{\texttt{sat_cjnt}\ s\ (\mathit{denv}, \mathit{env}\ \texttt{|+}\ (v, \texttt{evalex}\ \mathit{env}\ x), g)}{\texttt{sat_cjnt}\ s\ (\mathit{denv}, \mathit{env}, \texttt{LetVal1}\ (v, x)\ g)}$$

$$\frac{\texttt{sat_cjnt}\ s\ (\texttt{DUPDATE}\ \mathit{denv}\ \mathit{env}\ (\texttt{NonRec}\ \mathit{def}), \mathit{env}, g)}{\texttt{sat_cjnt}\ s\ (\mathit{denv}, \mathit{env}, \texttt{Let1}\ \mathit{def}\ g)}$$

$$\frac{\texttt{sat_cjnt}\ s\ (\texttt{DUPDATE}\ \mathit{denv}\ \mathit{env}\ (\texttt{Rec}\ (\texttt{FEMPTY}\ \texttt{|++}\ \mathit{defs})), \mathit{env}, g)}{\texttt{sat_cjnt}\ s\ (\mathit{denv}, \mathit{env}, \texttt{LetRec1}\ \mathit{defs}\ g)}$$

$$\frac{\begin{array}{c}\texttt{DLOOKUP}\ \mathit{denv}\ \texttt{"Eqv"} = \texttt{NONE} \\ s \triangleleft \texttt{evalex}\ \mathit{env}\ e_1 = s \triangleleft \texttt{evalex}\ \mathit{env}\ e_2\end{array}}{\texttt{sat_cjnt}\ s\ (\mathit{denv}, \mathit{env}, \texttt{Call}\ \texttt{"Eqv"}\ [e_1;\ e_2])}$$

$$\frac{\begin{array}{c}\texttt{DLOOKUP}\ \mathit{denv}\ \mathit{name} = \texttt{SOME}\ (\mathit{cdenv}, \mathit{cenv}, \texttt{Rec}\ d, \mathit{fmls}, \mathit{body}) \\ \texttt{sat_cjnt}\ s\ (\texttt{DUPDATE}\ \mathit{cdenv}\ \mathit{cenv}\ (\texttt{Rec}\ d), \texttt{call_env}\ \mathit{cenv}\ \mathit{fmls}\ (\texttt{evalex}\ \mathit{env})\ \mathit{args}, \mathit{body})\end{array}}{\texttt{sat_cjnt}\ s\ (\mathit{denv}, \mathit{env}, \texttt{Call}\ \mathit{name}\ \mathit{args})}$$

$$\frac{\begin{array}{c}\texttt{DLOOKUP}\ \mathit{denv}\ \mathit{name} = \texttt{SOME}\ (\mathit{cdenv}, \mathit{cenv}, \texttt{NonRec}\ d, \mathit{fmls}, \mathit{body}) \\ \texttt{sat_cjnt}\ s\ (\mathit{cdenv}, \texttt{call_env}\ \mathit{cenv}\ \mathit{fmls}\ (\texttt{evalex}\ \mathit{env})\ \mathit{args}, \mathit{body})\end{array}}{\texttt{sat_cjnt}\ s\ (\mathit{denv}, \mathit{env}, \texttt{Call}\ \mathit{name}\ \mathit{args})}$$

We now proceed to prove the lemmas towards the soundness theorem.

Lemma 42. *If a step from a state produces a substitution, the substitution satisfies the state*

$$\vdash \texttt{is_sbag}\ \mathit{df}\ \wedge\ \texttt{is_sbag}\ \mathit{cf}\ \wedge\ \texttt{wf_state}\ \mathit{df}\ \mathit{cf}\ \mathit{state}\ \wedge\ \texttt{step}\ \mathit{df}\ \mathit{cf}\ \mathit{state}\ (\texttt{SOME}\ s)\ \mathit{cont} \Rightarrow \texttt{sat_state}\ \mathit{df}\ \mathit{cf}\ s\ \mathit{state}$$

Proof. A substitution is only emitted when there are no conjuncts left in a disjunct. Therefore the disjunct that emitted the substitution is trivially satisfied. □

Lemma 43. *The state before a satisfied state is also satisfied*

$$\vdash \texttt{is_sbag}\ \mathit{df}\ \wedge\ \texttt{is_sbag}\ \mathit{cf}\ \wedge\ \texttt{wf_state}\ \mathit{df}\ \mathit{cf}\ \mathit{state}\ \wedge\ \texttt{step}\ \mathit{df}\ \mathit{cf}\ \mathit{state}\ \mathit{ans}\ \mathit{cont} \Rightarrow \texttt{sat_state}\ \mathit{df}\ \mathit{cf}\ \mathit{ss}\ \mathit{cont} \Rightarrow \texttt{sat_state}\ \mathit{df}\ \mathit{cf}\ \mathit{ss}\ \mathit{state}$$

Proof. If a satisfied disjunct in *state* is untouched by the transition to *cont*, then *cont* is immediately satisfied. So we assume we took a step on the sole satisfied disjunct in *state*, and do a case analysis on the type of conjunct that was stepped. In most cases the rules of `sat_cjnt` match the rules of `step`, and the result follows easily. In the case where the step was on a call to `"Eqv"`, we make use of Theorem 3. □

The next lemma does most of the work required for Lemma 45. We prove this lemma separately, because it can be stated about conjuncts in general, whereas Lemma 45 about initial states only.

Lemma 44. *If a conjunct is satisfied, its goal expressions is satisfied as a formula by the same substitution*

$$\vdash \texttt{sat_cjnt}\ s\ c \Rightarrow \forall d\ e\ g.\ c = (d,e,g)\ \wedge\ \texttt{wfs}\ s\ \wedge\ \texttt{wf_cjnt}\ c \Rightarrow \texttt{sat}\ (\texttt{denv_map}\ (\lambda e.\ \texttt{walk}^*\ s \circ e)\ \texttt{I}\ d)\ (\texttt{walk}^*\ s \circ e)\ g$$

Proof. By rule induction on `sat_cjnt`. □

Lemma 45. *If the initial machine state is satisfied, the corresponding program is satisfied*

$$\vdash \texttt{is_sbag}\ \mathit{df}\ \wedge\ \texttt{is_sbag}\ \mathit{cf}\ \wedge\ \texttt{wf_state}\ \mathit{df}\ \mathit{cf}\ \mathit{state}\ \wedge\ \texttt{sat_state}\ \mathit{df}\ \mathit{cf}\ s\ \mathit{state}\ \wedge\ \texttt{bag_of_sbag}\ \mathit{df}\ \mathit{state} = \{\!|(\texttt{FEMPTY},\mathit{cs})|\!\}\ \wedge\ \texttt{bag_of_sbag}\ \mathit{cf}\ \mathit{cs} = \{\!|(\texttt{DEMPTY},\texttt{FEMPTY}\ |\!+\ (q,\texttt{Var}\ n),g)|\!\} \Rightarrow \texttt{sat}\ \texttt{DEMPTY}\ (\texttt{FEMPTY}\ |\!+\ (q, s \lhd \texttt{Var}\ n))\ g$$

Proof. We use Lemma 44 on the sole conjunct in the initial state. Since the definition environment is empty, it us unaffected by being mapped over by $\texttt{walk}^*\ s$. □

Theorem 9. *Soundness of non-deterministic with respect to high-level semantics*

$\vdash$ is_sbag df $\wedge$ is_sbag cf $\wedge$ wf_state df cf $state$ $\wedge$
bag_of_sbag df $state$ = $\{\!|$(FEMPTY,cs)$|\!\}$ $\wedge$
bag_of_sbag cf cs = $\{\!|$(DEMPTY,FEMPTY |+ (q,Var n),g)$|\!\}$ $\wedge$
stream_of (step df cf) $state$ $llist$ $\Rightarrow$
every
($\lambda\, a.\ \forall s.\ a$ = SOME s $\Rightarrow$ sat DEMPTY (FEMPTY |+ (q,s $\triangleleft$ Var n)) g)
$llist$

Proof. By coinduction on every, using the three lemmas above. □

5.6 Executable semantics

The implementation of miniKanren in Scheme looks considerably different from the abstract machine we considered in the last section. The Scheme implementation uses a concept of a *goal*, which is represented by a function that takes a substitution and returns a possibly infinite stream of substitutions. Every goal expression in our syntax gives rise to such a goal. More accurately, every conjunct, as defined in the previous section, represents a goal, since we need environments binding the free variables and relations in the goal expression, The stream of answer substitutions in the Scheme implementation is obtained by operating on and combining the streams returned by calls to goals.

The streams returned by goals are combined in an interleaving fashion, although it is not necessarily perfect interleaving: elements from one stream are not always separated by elements from the other in the combined stream. Definition 46 gives a HOL data type, (α, β) inf, to model an interleaving stream that includes the goal expression (plus environments) that will generate the rest of the stream if necessary. The type variable α represents elements of the stream, which we would usually take to be substitutions. The type variable β is the type of constants in our expressions. The interleaving streams data type presented here is related to the backtracking monad transformer described in [Kiselyov et al. 2005].

Definition 46. Interleaving streams

```
(α, β) inf
  = Mzero
  | Inc of (α, β) f
  | Apply of (α, β) f
  | Sing of α
  | ConsInf of α => (α, β) f ;

(α, β) f
  = Mplus of (α, β) inf => (α, β) f
  | Bind of (α, β) inf => β conjunct
  | Goal of α => β conjunct
```

The type (α, β) `inf` is defined in mutual recursion with another type (α, β) `f`. The intuition behind these types is that the `inf` type represents a stream and the `f` type represents a suspended computation that will produce a stream. Let us describe the constructors in the `inf` type. `Mzero` represents the empty stream. `Inc` f is a suspended computation to generate a stream, which we can treat as a stream. `Apply` f represents a stream-generating computation that has been unsuspended. The difference between `Apply` f and `Inc` f is that elements of the latter are delayed longer than elements of the former. This affects the order of elements when two streams are combined. `Sing` s is a stream that ends after a single element s. `ConsInf` s f is a stream with an element s at the front and of which f generates the rest.

Now we describe the constructors of the `f` type. `Mplus` *inf* f represents a combination of the streams *inf* and `Inc` f. `Bind` *inf* (d,e,g) represents the combination of all the streams obtained by applying the goal (d,e,g) to elements of the stream *inf*. Finally `Goal` s (d,e,g) is the stream obtained by applying the goal (d,e,g) to the element s. As an example of these ideas, we might have defined `Sing` s = `ConsInf` s `(Bind Mzero ARB)`: a `Bind` of `Mzero` to any goal is empty, since there are no elements to apply the goal to. We use a separate constructor for singletons to match the implementation of miniKanren, where it enables better memory usage.

So much for the data types. To make these descriptions of the constructors accurate, we have a function `stepinf` that takes an `inf` and can be unfolded to a HOL lazy list. The `inf` streams are "stepped" at a finer granularity than the sequence of elements they contain; the intuition is that it may take a number of computational steps to retrieve the next element. The lazy list obtained by unfolding `stepinf` contains elements `SOME` a, where a is an element of the stream, and `NONE` indicating that a step was taken but it didn't produce an element.

`stepinf` is defined in mutual recursion with `stepf` (Figure 5.6 on page 78). The first argument *sg* is a function used to step a `Goal` in the last line of `stepf`. We keep *sg* abstract for the moment, because `stepinf` and `stepf` animate the interleaving behavior of our streams data types independently of how goals are stepped. Since `stepinf` is designed to be unfolded, it returns `NONE` when the stream being stepped is empty, otherwise it returns `SOME` (*inf*,*res*) where *inf* is the stream after a step and *res* is the result of the step (either a stream element or `NONE`). `ConsInf` and `Sing` streams have a known element a at the front, so this is used as the result; the other constructors have a `NONE` result.

The `stepf` function does a step on a stream-producing computation, and returns a stream. The clauses of `stepf` were derived from the `mplus` and `bind` functions in the implementation of miniKanren, which capture the interleaving search strategy.

Figure 5.6 on page 79 gives the function, `stepgoal`, that we use as the *sg* argument of `stepinf`. Most of the clauses are similar to those of `step` in the previous section. There is new material in the treatment of conjunction and disjunction using streams. The result of conjoining two goals is obtained by feeding the stream of answers from the first goal into the second and interleaving all the streams produced by the second. This is accomplished by returning a `Bind` stream. The result of a disjunction is obtained by interleaving the results of the two goals using `Mplus`.

Definition 47. Stepping interleaving streams

```
stepinf sg (ConsInf a f) = SOME (Inc f,SOME a)
stepinf sg (Sing a) = SOME (Mzero,SOME a)
stepinf sg (Apply f) = SOME (stepf sg f,NONE)
stepinf sg (Inc f) = SOME (stepf sg f,NONE)
stepinf sg Mzero = NONE

stepf sg (Mplus Mzero f) = Apply f
stepf sg (Mplus (Apply f') f) = Apply (Mplus (stepf sg f') f)
stepf sg (Mplus (Inc f') f) = Inc (Mplus (Apply f) f')
stepf sg (Mplus (Sing a) f) = ConsInf a f
stepf sg (Mplus (ConsInf a f') f) = ConsInf a (Mplus (Apply f) f')
stepf sg (Bind Mzero c) = Mzero
stepf sg (Bind (Apply f) c) = Apply (Bind (stepf sg f) c)
stepf sg (Bind (Inc f) c) = Inc (Bind (Apply f) c)
stepf sg (Bind (Sing a) c) = Apply (Goal a c)
stepf sg (Bind (ConsInf a f) c) =
  Apply (Mplus (Apply (Goal a c)) (Bind (Apply f) c))
stepf sg (Goal a c) = sg a c
```

Figure 5.4: Functions to perform one computational step on our interleaving streams type.

We can also see a use of having separate recursive and non-recursive definition types in `stepgoal`. In the `Call` clause, we use the `Inc` stream constructor for a recursive goal, but `Apply` for a non-recursive goal. This means that only recursive goals will trigger interleaving, which is reasonable since only recursive goals can lead to infinitely many steps that would drown out other sources of answers. We also see the use of the `Sing` constructor to create the single element stream that is the result of a unification goal.

Our model of potentially infinite streams, consisting of `stepinf` and `stepf` and the two data types, is essentially a "defunctionalisation" [Reynolds 1998] of the Scheme implementation, replacing procedure application with the `Apply` constructor. This transformation preserves executability. We are able to emit the definitions from this section as SML code, and run the resulting implementation on the test suite used for the Scheme implementation of miniKanren (with a minor translation of the tests from Scheme to SML). All the tests pass, and the SML implementation is comparably efficient to the Scheme one. Importantly, since `stepgoal` calls `unify`, this is also a test of the executability of our definition from Chapter 3.

Definition 48. One step on a goal

```
stepgoal a (d,e,Conj g1 g2) =
  Apply (Bind (Apply (Goal a (d,e,g1))) (d,e,g2)) ∧
stepgoal a (d,e,Disj g1 g2) =
  Apply (Mplus (Apply (Goal a (d,e,g1))) (Goal a (d,e,g2))) ∧
stepgoal (s,n) (d,e,LetVar v g) =
  Apply (Goal (s,n + 1) (d,e |+ (v,Var n),g)) ∧
stepgoal (s,n) (d,e,LetVal1 (v,t) g) =
  Apply (Goal (s,n) (d,e |+ (v,evalex e t),g)) ∧
stepgoal a (d,e,LetRec1 ls g) =
  Apply (Goal a (DUPDATE d e (Rec (FEMPTY |++ ls)),e,g)) ∧
stepgoal a (d,e,Let1 nfb g) =
  Apply (Goal a (DUPDATE d e (NonRec nfb),e,g)) ∧
stepgoal a (d,e,Call name args) =
  case DLOOKUP d name of
    SOME (cd,ce,defs,fmls,g) →
      (let ce′ = call_env ce fmls (evalex e) args in
         case defs of
           Rec _ → Inc (Goal a (DUPDATE cd ce defs,ce′,g))
         ‖ NonRec _ → Apply (Goal a (cd,ce′,g)))
  ‖ NONE →
      case (name,args) of
        ("Eqv",[e1; e2]) →
          case unify (FST a) (evalex e e1) (evalex e e2) of
            NONE → Mzero
          ‖ SOME sx → Sing (sx,SND a)
```

Figure 5.5: Function to perform one step of applying a goal to a substitution a.

Chapter 6

Conclusion

6.1 The import of this work

This thesis describes mechanically verified theories both of unification with triangular substitutions and of the semantics of miniKanren. These theories put the miniKanren approach to logic programming on a good theoretical foundation.

The fact that the work was done mechanically gives a higher level of assurance: our theoretical foundations are sounder, and we are much more confident that our implementations are correct. Mechanisation forces thoroughness: for example, well-formedness of triangular substitutions may have been overlooked as "too obvious" in a less rigorous setting, but is crucial to the termination of `walk*` and therefore well-formedness preservation must be one of the correctness criteria for `unify`.

Unification is a well-studied problem, and theories of substitution and unification have been developed in various machine-checked formal logics. The novelty of our unification theories is twofold. First, we have mechanised algorithms in accumulator-passing style. This style adds to the complexity of the verification, but helps to make the algorithms directly implementable (*i.e.*, deterministic, with suitably concrete data structures), and efficient, as our tests demonstrate. Second, our algorithms use triangular substitutions, which haven't yet received much formal treatment.

miniKanren is a pure logic programming language designed for pedagogy and research. The theories in this thesis are the first development of the formal side of miniKanren. The various implementations of miniKanren explore areas of LP that we haven't considered here. Extending our formal theories will enable a level of assessment of these research areas above testing.

6.2 Related work

6.2.1 First-order unification

We concentrate on machine-checked work since this is closest to ours. One of the first mechanical verifications of a unification algorithm was done by Larry Paulson [Paulson 1985] in the theorem prover LCF, a predecessor of HOL4. The algorithm verified is the `Iunify` algorithm we saw in Section 3.6. At the time, Paulson had to work through justifying a definition *via* a well-founded relation manually. Mechanical verification of

the same algorithm (in HOL) was used as an example in [Slind 1999] of Slind's library TFL for defining terminating functions. As with all the other mechanisations we are aware of, `Iunify` returns idempotent substitutions, but it is a directly implementable algorithm.

The most sophisticated mechanisation of unification is [Ruiz-Reina et al. 2006], which presents a formalisation of a quadratic-time unification algorithm in the first-order theorem prover ACL2. The ACL2 system is also a programming language, and the verified algorithm can be executed directly. The basic approach of this work is to develop theory about the non-deterministic transformation system (MMTS, see Section 3.6). This is then refined into a deterministic strategy that also uses term-graphs, ensuring quadratic performance.

Another mechanical verification is described in [McBride 2003]. This work was performed in Lego, and makes heavy use of that system's dependent types. This technology allows for a simple termination argument, once one has established that the function definition is indeed well-typed. McBride also includes a list of some earlier mechanisations.

A recent paper [de Moura et al. 2010] describes work mechanising Robinson's original unification algorithm [Robinson 1965] in PVS. This algorithm is of historical interest, but would perform very badly if implemented.

6.2.2 Nominal unification

The only mechanised treatment of nominal unification is [Urban 2004], which is based on the journal presentation [Urban et al. 2004]. Their algorithm is a non-deterministic transformation system, in the established MMTS style. The implementation of αKanren's unification algorithm originally used idempotent substitutions, and was directly based on the transformation system. The algorithm mechanised in Chapter 4 (`nomunify`) is a new version that was rewritten to use triangular substitutions and to better match the style of miniKanren's algorithm (`unify`).

There have been a number of improvements to the efficiency of nominal unification, typically using techniques borrowed from first-order unification. See, for example, [Calvès and Fernández 2008] and [Levy and Villaret 2008]. To date, none of this work has been mechanised.

6.2.3 Semantics of logic programming languages

There are many approaches to giving semantics to logic programming, and indeed many logic programming languages, but we won't mention them all here. A standard reference in this area is [Lloyd 1987], which gives the theory for the pure part of the "canonical" LP language, Prolog. This is similar to our high-level semantics in Section 5.3.

A reasonably widespread approach to the operational semantics of Prolog is the Warren Abstract Machine (WAM) [Warren 1983; Aït-Kaci 1991]. This semantics was designed to enable research on Prolog compilers and has been used to investigate com-

piler optimisations. The WAM is a good example of how a semantics can provide a unifying framework of concepts to enable directed research.

Our executable semantics is based on the interleaving streams described in [Kiselyov et al. 2005]. The use of suspended computations to create potentially infinite streams is a staple idea in functional programming, going back at least as far as [Friedman and Wise 1976]. An example of work on using streams as the basis for search in logic programming is [Spivey 2000].

6.3 Future work

We have verified the four main results—termination, soundness, generality, and completeness—for both the first-order and nominal unification algorithms used in miniKanren and αKanren. One way to extend this work is to formally verify the complexity of our unification algorithms. For time complexity, this might involve adding an extra argument that accumulates the number of steps taken during each recursive call. Adding such an argument is eased by the fact that our algorithms are already in accumulator-passing style.

Two other ways to continue the unification theories are to mechanically verify more efficient approaches to unification, and to see whether they can be adapted to produce triangular substitutions. The ACL2 verification in [Ruiz-Reina et al. 2006] is of a quadratic algorithm; the linear algorithm in [Martelli and Montanari 1982] could be given a similar treatment. The best way to approach an efficient triangular substitution algorithm may be to try adapting the non-deterministic transformation system to work with triangular substitutions first. Then a policy of deterministic choices could be overlaid on this triangular system, as was done in the ACL2 verification for the idempotent system.

The condition on substitutions to make `vwalk_rhs` in Section 2.5.1 a suitable replacement for `vwalk_al` should be worked out, so we can confirm that `unify` preserves it. There may be a number of interesting conditions that `unify` preserves. We could extend our theory of triangular substitutions by finding more predicates, apart from well-formedness, for classifying triangular substitutions.

There has been some unpublished work on concrete representations for substitutions in miniKanren that are more sophisticated than association lists. Our theories are robust to changes in concrete representation, since we only depend on the behavior of substitutions as finite maps. However, mechanisation of any theory on maintaining more sophisticated representations—their efficiency and their correctness as finite maps—would complement our work so far on triangular representations. A good source of inspiration for efficient, persistent data structures is [Okasaki 1998]. Additionally, [Kaplan 2004] provides a good introduction to persistence.

The main result in Chapter 5, soundness of operational with respect to logical semantics, is but one of the three results we would initially ask of a semantics. As mentioned in that section, the other two are completeness and fairness of the operational semantics. From initial attempts at the completeness result, it seems that the lexically scoped existential variables in miniKanren's syntax bring some difficulties. In

particular, the invariants in Definition 45 need to be extended to keep track of variables that are already used in a disjunct and so can't be introduced again by `LetVar`. Aside from this complication, the main difficulty for the completeness is that a final answer doesn't specify what the intermediate variables, as introduced by `LetVar`, should be bound to. It only says what the query variable should ultimately be bound to, and there is freedom to go through any number of intermediate bindings.

It should be reasonably straightforward to show that an expensive search strategy like breadth-first search is fair. Breadth-first search can be represented by a structured bag with a queue-like replace operation. The difficulty of proving fairness for more complicated search strategies will likely depend on the strategy.

Our operational semantics is not executable until a search strategy is fixed. The search strategy we are most interested in is the one miniKanren actually uses, which we have captured in the executable semantics. What is missing is a formal link between the executable and operational semantics. This could be achieved by somehow interpreting elements of the `(`α`, `β`) inf` type as machine states, that is, as containing dynamic conjuncts and disjuncts. Then we would need to determine how `stepinf` and `stepf` can be considered as a structured bag replacement relation. The advantage of establishing a correspondence between the executable and operational semantics is that correctness properties (soundness and completeness) of the operational semantics will automatically transfer to the executable semantics. They are presumably easier to prove for the non-deterministic abstract machine; soundness certainly was.

miniKanren has been extended in a number of directions for the sake of research on pure logic programming. The extension to nominal logic in αKanren is an example. Other extensions include: the addition of *disequality constraints*, which means providing a $\not\equiv$ relation; and the addition of *tabled* relations, which means caching answers from relations both for efficiency and to avoid divergence when the same answer is repeated infinitely many times. Another aspect of miniKanren is *reification* of answer terms, which means picking canonical representatives from classes of equivalent answers, and which is an open problem in the nominal case. All of these are described further in [Byrd 2009]. A natural extension of our semantics for core miniKanren would be to cover these extended language features.

Appendix A

List of Theorems in Mechanised Theories

We use this appendix to give statements of all the theorems in a selection of the theories developed during the formalisation of miniKanren presented in the main text. Many of these are the "glue" theorems mentioned in Section 1.4. The main reason to list them is to give an idea of the proportion of theorems that made it into the main text.

Each section of the appendix is named after a HOL4 theory developed during the Honours year, and contains the definitions and theorems proved in that theory. There are 20 theories in all, and we list the theorems in 9 of them. The theorems and theories are presented in a topologically sorted order: if one theorem depends on another for its proof (and both are listed), then the former will be listed below the latter.

We don't list the induction theorems of non-primitive recursive functions. For definitions of inductive relations, we give the theorem specifying the rules, but not the induction and cases theorems; also, we display the rules theorem as a conjunction of implications rather than using the horizontal line style of, for example, Definition 29.

A.1 term

1. `datatype_term` (Definition 1)

   ```
   α term = Var of num | Pair of α term => α term | Const of α
   ```

2. `pair_count_def`

   ```
   pair_count (Var v) = 0
   pair_count (Const c) = 0
   pair_count ⟨t1, t2⟩ = 1 + pair_count t1 + pair_count t2
   ```

3. `vars_def` (Definition 2)

   ```
   vars (Var x) = {x}
   vars ⟨t1, t2⟩ = vars t1 ∪ vars t2
   vars (Const v0) = { }
   ```

4. FINITE_vars

$\vdash$ FINITE (vars t)

5. varb_def

varb (Var s) = $\{\!|s|\!\}$
varb $\langle t_1, t_2\rangle$ = varb t_1 $\uplus$ varb t_2
varb (Const c) = $\{\!|\,|\!\}$

6. FINITE_varb

$\vdash$ FINITE_BAG (varb t)

7. IN_varb_vars

$\vdash$ e $\in$: varb t $\iff$ $e \in$ vars t

A.2 subst

8. rangevars_def

rangevars s = BIGUNION (IMAGE vars (FRANGE s))

9. FINITE_rangevars

$\vdash$ FINITE (rangevars s)

10. IN_FRANGE_rangevars

$\vdash$ $t \in$ FRANGE $s \Rightarrow$ vars $t \subseteq$ rangevars s

11. rangevars_FUPDATE

$\vdash$ $v \notin$ FDOM $s \Rightarrow$ rangevars (s |+ (v,t)) = rangevars $s \cup$ vars t

12. substvars_def

substvars s = FDOM $s \cup$ rangevars s

13. FINITE_substvars

$\vdash$ FINITE (substvars s)

14. vR_def (Definition 7)

$\mathtt{tri}_{\mathtt{R}}$ s y x $\iff$ case s ' x of NONE $\to$ F $\|$ SOME $t \to y \in$ vars t

15. wfs_def

wfs s $\iff$ WF ($\mathtt{tri}_{\mathtt{R}}$ s)

16. `wfs_FEMPTY`

$\vdash$ `wfs FEMPTY`

17. `wfs_SUBMAP`

$\vdash$ `wfs` $sx \wedge s \sqsubseteq sx \Rightarrow$ `wfs` s

18. `wfs_no_cycles` (Lemma 3)

$\vdash$ `wfs` $s \iff \forall v.\ \neg(\texttt{tri}_{\texttt{R}}\ s)^{+}\ v\ v$

19. `subst_APPLY_def` (Definition 3)

s `"` `Var` v = `case` s `'` v `of` `NONE` $\rightarrow$ `Var` v $\|$ `SOME` $t \rightarrow t$
s `"` $\langle t_1, t_2\rangle = \langle s$ `"` t_1, s `"` $t_2\rangle$
s `"` `Const` c = `Const` c

20. `subst_APPLY_FAPPLY`

$\vdash v \in$ `FDOM` $s \Rightarrow s$ `'` $v = s$ `"` `Var` v

21. `noids_def` (Definition 4)

`noids` $s \iff \forall v.\ s$ `'` $v \neq$ `SOME (Var` v`)`

22. `subst_APPLY_id`

$\vdash s$ `"` $t = t \iff \forall v.\ v \in$ `vars` $t \wedge v \in$ `FDOM` $s \Rightarrow s$ `'` $v =$ `Var` v

23. `idempotent_def` (Definition 5)

`idempotent` $s \iff \forall t.\ s$ `"` s `"` $t = s$ `"` t

24. `wfs_noids` (Corollary 1)

$\vdash$ `wfs` $s \Rightarrow$ `noids` s

25. `idempotent_rangevars` (Lemma 1)

$\vdash$ `idempotent` $s \wedge$ `noids` $s \iff$ `DISJOINT (FDOM` s`) (rangevars` s`)`

26. `wfs_FAPPLY_var`

$\vdash$ `wfs` $s \Rightarrow \forall v.\ v \in$ `FDOM` $s \Rightarrow s$ `'` $v \neq$ `Var` v

27. `TC_vR_vars_FRANGE`

$\vdash (\texttt{tri}_{\texttt{R}}\ s)^{+}\ u\ v \Rightarrow v \in$ `FDOM` $s \Rightarrow u \in$ `BIGUNION (IMAGE vars (FRANGE` s`))`

28. `wfs_idempotent` (Corollary 2)

$\vdash$ `idempotent` $s \wedge$ `noids` $s \Rightarrow$ `wfs` s

29. `selfapp_def` (see Definition 6 on page 22)

30. `selfapp_eq_iter_APPLY` (Lemma 2)

$\vdash$ `selfapp` s `"` t = s `"` s `"` t

31. `FDOM_selfapp`

$\vdash$ `FDOM (selfapp` s`)` = `FDOM` s

32. `IN_vars_APPLY`

$\vdash$ $v \in$ `vars` $(s$ `"` $t) \iff$
$v \notin$ `FDOM` $s \land v \in$ `vars` $t \lor \exists x.$ $\mathtt{tri}_{\mathtt{R}}$ s v $x \land x \in$ `vars` t

33. `vR1_def` (Definition 8)

$\mathtt{tri}^1_{\mathtt{R}}$ s y $x \iff \mathtt{tri}_{\mathtt{R}}$ s y $x \land y \notin$ `FDOM` s

34. `vR_selfapp`

$\vdash$ $\mathtt{tri}_{\mathtt{R}}$ `(selfapp` s`)` = $\mathtt{tri}^1_{\mathtt{R}}$ s `RUNION NRC` $(\mathtt{tri}_{\mathtt{R}}$ $s)$ `2`

35. `vR1_selfapp`

$\vdash$ $\mathtt{tri}^1_{\mathtt{R}}$ `(selfapp` s`)` = $\mathtt{tri}^1_{\mathtt{R}}$ s `RUNION` $\mathtt{tri}_{\mathtt{R}}$ s `O` $\mathtt{tri}^1_{\mathtt{R}}$ s

36. `FDOM_FUNPOW_selfapp`

$\vdash$ `FDOM` $(\mathtt{selfapp}^n$ $s)$ = `FDOM` s

37. `NRC_2_IMP_TC_vR_selfapp`

$\vdash$ `NRC` $(\mathtt{tri}_{\mathtt{R}}$ $s)$ `(2 * SUC` n`)` v $u \Rightarrow (\mathtt{tri}_{\mathtt{R}}$ `(selfapp` s`))`$^+$ v u

38. `NRC_2_1_IMP_TC_vR_selfapp`

$\vdash$ `NRC` $(\mathtt{tri}_{\mathtt{R}}$ $s)$ `(2 *` n`)` v $u \land \mathtt{tri}^1_{\mathtt{R}}$ s w $v \Rightarrow (\mathtt{tri}_{\mathtt{R}}$ `(selfapp` s`))`$^+$ w u

39. `TC_vR_selfapp` (Lemma 4)

$\vdash$ $(\mathtt{tri}_{\mathtt{R}}$ `(selfapp` s`))`$^+$ v $u \iff$
$(\exists n.$ `NRC` $(\mathtt{tri}_{\mathtt{R}}$ $s)$ `(2 * SUC` n`)` v $u) \lor$
$\exists n$ $u'.$ `NRC` $(\mathtt{tri}_{\mathtt{R}}$ $s)$ `(2 *` n`)` u' $u \land \mathtt{tri}^1_{\mathtt{R}}$ s v u'

40. `wfs_selfapp` (Lemma 5)

$\vdash$ `wfs` $s \iff$ `wfs (selfapp` s`)`

41. `vR_LRC_ALL_DISTINCT`

$\vdash$ `wfs` $s \Rightarrow \forall ls$ v $u.$ `LRC` $(\mathtt{tri}_{\mathtt{R}}$ $s)$ ls v $u \Rightarrow$ `ALL_DISTINCT` ls

42. `vR_LRC_FDOM`

$\vdash$ `LRC` $(\mathtt{tri}_{\mathtt{R}}$ $s)$ $(h$`::`$t)$ v $u \land$ `MEM` e $t \Rightarrow e \in$ `FDOM` s

43. vR_LRC_bound

$\vdash$ wfs s $\wedge$ LRC (tri$_{\mathrm{R}}$ s) ls v u $\Rightarrow$ LENGTH ls $\leq$ CARD (FDOM s) + 1

44. idempotent_selfapp

$\vdash$ idempotent s $\iff$ selfapp s = s

45. fixpoint_IMP_wfs

$\vdash$ idempotent (selfapp^{n} s) $\wedge$ noids (selfapp^{n} s) $\Rightarrow$ wfs s

46. idempotent_substeq

$\vdash$ (") s_1 = (") s_2 $\Rightarrow$ (idempotent s_1 $\iff$ idempotent s_2)

47. vR_FUNPOW_selfapp_bound

$\vdash$ tri$_{\mathrm{R}}$ (selfapp^{n} s) v u $\Rightarrow$
  $\exists\, m$. 1 $\leq$ m $\wedge$ NRC (tri$_{\mathrm{R}}$ s) m v u $\wedge$ (v $\in$ FDOM s $\Rightarrow$ n $\leq$ m)

48. idempotent_or_vR

$\vdash$ idempotent s $\vee$ $\exists\, u\ v$. tri$_{\mathrm{R}}$ s v u $\wedge$ v $\in$ FDOM s

49. wfs_IMP_fixpoint

$\vdash$ wfs s $\Rightarrow$ $\exists\, n$. idempotent (selfapp^{n} s) $\wedge$ noids (selfapp^{n} s)

50. wfs_iff_fixpoint (Theorem 1)

$\vdash$ wfs s $\iff$ $\exists\, n$. idempotent (selfapp^{n} s) $\wedge$ noids (selfapp^{n} s)

A.3 collapse

51. collapse_def (Definition 12)

collapse s = FUN_FMAP ($\lambda\, v$. s $\triangleleft$ Var v) (FDOM s)

52. collapse_APPLY_eq_walkstar (Lemma 10)

$\vdash$ wfs s $\Rightarrow$ $\forall\, t$. collapse s " t = s $\triangleleft$ t

53. collapse_FAPPLY_eq_walkstar

$\vdash$ wfs s $\Rightarrow$ $\forall\, v$. v $\in$ FDOM s $\Rightarrow$ collapse s ' v = s $\triangleleft$ Var v

54. walkstar_unchanged

$\vdash$ wfs s $\Rightarrow$ $\forall\, t$. DISJOINT (vars t) (FDOM s) $\Rightarrow$ s $\triangleleft$ t = t

55. collapse_idempotent (Lemma 11)

$\vdash$ wfs s $\Rightarrow$ idempotent (collapse s)

56. idempotent_collapse (Lemma 12)

⊢ idempotent s ∧ noids s ⇒ collapse s = s

57. walkstar_eq_idem_APPLY (Lemma 13)

⊢ wfs s ⇒ (idempotent s ⟺ walk* s = (") s)

58. subst_APPLY_walkstar

⊢ wfs s ⇒ ∀ t. s " s ◁ t = s ◁ t

A.4 walk

59. vwalk_def (Definition 9)

```
wfs s ⇒
  vwalk s v =
    case s ' v of
      SOME (Var u) → vwalk s u
    ‖ SOME t → t
    ‖ NONE → Var v
```

60. NOT_FDOM_vwalk

⊢ wfs s ∧ v ∉ FDOM s ⇒ vwalk s v = Var v

61. vwalk_to_var

⊢ wfs s ⇒ ∀ v u. vwalk s v = Var u ⇒ u ∉ FDOM s

62. walk_def

```
walk s t =
  case t of
    Var v → vwalk s v
  ‖ ⟨v6, v7⟩ → ⟨v6, v7⟩
  ‖ Const v8 → Const v8
```

63. walk_thm

⊢ walk s (Var v) = vwalk s v ∧ walk s $\langle t_1, t_2\rangle$ = $\langle t_1, t_2\rangle$ ∧
walk s (Const c) = Const c

64. walk_FEMPTY

⊢ walk FEMPTY t = t ∧ vwalk FEMPTY v = Var v

65. walk_var_vwalk

⊢ wfs s ⇒ walk s t = Var v ⇒ ∃ u. t = Var u ∧ vwalk s u = Var v

66. `walk_to_var`

$\vdash$ `wfs` s $\wedge$ `walk` s t = `Var` v $\Rightarrow$ $v \notin$ `FDOM` s $\wedge$ $\exists u.$ t = `Var` u

67. `vwalk_vR`

$\vdash$ `wfs` s $\Rightarrow$
  $\forall v$ u. $u \in$ `vars (vwalk` s v`)` $\wedge$ `vwalk` s $v \neq$ `Var` v $\Rightarrow$ `(tri`$_{\mathtt{R}}$ s`)`$^{+}$ u v

68. `vwalk_IN_FRANGE`

$\vdash$ `wfs` s $\wedge$ $v \in$ `FDOM` s $\Rightarrow$ `vwalk` s $v \in$ `FRANGE` s

69. `walk_IN_FRANGE`

$\vdash$ `wfs` s $\wedge$ `walk` s $t \neq t$ $\Rightarrow$ `walk` s $t \in$ `FRANGE` s

70. `vwalk_SUBMAP`

$\vdash$ `wfs` sx $\Rightarrow$
  $\forall v$ s.
    $s \sqsubseteq sx$ $\Rightarrow$
      case `vwalk` s v of
        `Var` u $\rightarrow$ `vwalk` sx v = `vwalk` sx u
      $\|$ $\langle v_6, v_7 \rangle$ $\rightarrow$ `vwalk` sx v = $\langle v_6, v_7 \rangle$
      $\|$ `Const` v_8 $\rightarrow$ `vwalk` sx v = `Const` v_8

71. `vwalk_al_thm`

$\vdash$ `wfs (alist_to_fmap` al`)` $\Rightarrow$
  `vwalk_al` al v =
    case `ALOOKUP` al v of
      `NONE` $\rightarrow$ `Var` v
    $\|$ `SOME (Var` u`)` $\rightarrow$ `vwalk_al` al u
    $\|$ `SOME` $\langle v_7, v_8 \rangle$ $\rightarrow$ $\langle v_7, v_8 \rangle$
    $\|$ `SOME (Const` v_9`)` $\rightarrow$ `Const` v_9

72. `vwalk_al_eq_vwalk` (Lemma 16)

$\vdash$ `vwalk` s = `vwalk_al (fmap_to_alist` s`)`

73. `ELENGTH_def` (Definition 17)

`ELENGTH (`v`,[])` = `0`
`ELENGTH (`v`,`h`::`t`)` = if `SND` h = `Var` v then `0` else `1 + ELENGTH (`v`,`t`)`

74. `vwalk_rhs_def` (see Definition 16 on page 33)

A.5 walkstar

75. `walkstar_thWF` (Lemma 7)

$\vdash \texttt{wfs}\ s \Rightarrow \texttt{WF}\ (\texttt{walk}^*{}_{\texttt{TR}}\ s)$

76. `walkstar_th1` (Lemma 8)

$\vdash \texttt{wfs}\ s \Rightarrow \forall t\ t_1\ t_2.\ \texttt{walk}\ s\ t = \langle t_1, t_2 \rangle \Rightarrow \texttt{walk}^*{}_{\texttt{TR}}\ s\ t_1\ t$

77. `walkstar_th2`

$\vdash \texttt{wfs}\ s \Rightarrow \forall t\ t_1\ t_2.\ \texttt{walk}\ s\ t = \langle t_1, t_2 \rangle \Rightarrow \texttt{walk}^*{}_{\texttt{TR}}\ s\ t_2\ t$

78. `walkstar_def` (Definition 10)

$$\begin{array}{l}\texttt{wfs}\ s \Rightarrow \\ \quad s \lhd t = \textsf{case}\ \texttt{walk}\ s\ t\ \textsf{of}\ \langle t_1, t_2 \rangle \rightarrow \langle s \lhd t_1, s \lhd t_2 \rangle \parallel t' \rightarrow t'\end{array}$$

79. `walkstar_FEMPTY`

$\vdash \texttt{FEMPTY} \lhd t = t$

80. `NOT_FDOM_walkstar`

$\vdash \texttt{wfs}\ s \Rightarrow \forall t.\ v \notin \texttt{FDOM}\ s \Rightarrow v \in \texttt{vars}\ t \Rightarrow v \in \texttt{vars}\ (s \lhd t)$

81. `TC_vR_vars_walkstar`

$\vdash \texttt{wfs}\ s \land u \in \texttt{vars}\ t \land (\texttt{tri}_{\texttt{R}}\ s)^+\ v\ u \land v \notin \texttt{FDOM}\ s \Rightarrow v \in \texttt{vars}\ (s \lhd t)$

82. `walkstar_SUBMAP` (Lemma 9)

$\vdash s \sqsubseteq sx \land \texttt{wfs}\ sx \Rightarrow sx \lhd t = sx \lhd s \lhd t$

83. `walkstar_idempotent`

$\vdash \texttt{wfs}\ s \Rightarrow \forall t.\ s \lhd t = s \lhd s \lhd t$

84. `walkstar_subterm_idem`

$\vdash s \lhd t_1 = \langle t_{11}, t_{12} \rangle \land \texttt{wfs}\ s \Rightarrow s \lhd t_{11} = t_{11} \land s \lhd t_{12} = t_{12}$

85. `walkstar_walk`

$\vdash \texttt{wfs}\ s \Rightarrow s \lhd \texttt{walk}\ s\ t = s \lhd t$

86. `walkstar_to_var`

$\vdash s \lhd t = \texttt{Var}\ v \land \texttt{wfs}\ s \Rightarrow v \notin \texttt{FDOM}\ s \land \exists u.\ t = \texttt{Var}\ u$

87. `apply_ts_thm` (see Lemma 6 on page 28)

88. `vars_walkstar`

$\vdash$ `wfs` $s \Rightarrow \forall t.$ `vars` $(s \triangleleft t) \subseteq$ `vars` $t \cup$ `BIGUNION (FRANGE (vars` $\circ$ s`))`

89. `oc_rules` (Definition 19)

$v \in$ `vars` $t \wedge v \notin$ `FDOM` $s \Rightarrow$ `oc` $s\ t\ v$
$u \in$ `vars` $t \wedge$ `vwalk` $s\ u = t' \wedge$ `oc` $s\ t'\ v \Rightarrow$ `oc` $s\ t\ v$

90. `oc_thm`

```
⊢ (wfs s ⇒
     (oc s (Var v) x  ⟺
        case vwalk s v of
          Var u → x = u
        ‖ ⟨t1, t2⟩ → oc s t1 x ∨ oc s t2 x
        ‖ Const v7 → F)) ∧
  (oc s ⟨t1, t2⟩ v  ⟺  oc s t1 v ∨ oc s t2 v) ∧ ¬oc s (Const c) v
```

91. `oc_walking` (Lemma 17)

```
⊢ wfs s ⇒
    (oc s t v  ⟺
       case walk s t of
         Var u → v = u
       ‖ ⟨t1, t2⟩ → oc s t1 v ∨ oc s t2 v
       ‖ Const v8 → F)
```

92. `oc_NOTIN_FDOM`

$\vdash$ `wfs` $s \Rightarrow \forall t\ v.$ `oc` $s\ t\ v \Rightarrow v \notin$ `FDOM` s

93. `oc_eq_vars_walkstar` (Lemma 18)

$\vdash$ `wfs` $s \Rightarrow$ `(oc` $s\ t\ v \iff v \in$ `vars` $(s \triangleleft t)$`)`

A.6 unifDef

94. `extension_chain`

```
⊢ FINITE (source z) ∧ (∀m n. m < n ⇒ subst m ⊑ subst n) ∧
  (∀n. DISJOINT (FDOM (subst n)) (source n)) ∧
  (∀n.
     z < n ⇒
       FDOM (subst n) ∪ source n = FDOM (subst z) ∪ source z) ⇒
    ∃x. ∀n. x < n ⇒ subst n = subst x
```

95. `wfs_extend`

$\vdash$ `wfs` $s \wedge v \notin$ `FDOM` $s \wedge v \notin$ `vars` $(s \triangleleft t) \Rightarrow$ `wfs` $(s$ `|+` $(v,t))$

96. `vwalk_FDOM`

$\vdash$ `wfs` s $\Rightarrow$
  `vwalk` s v $=$ t $\Rightarrow$
    $v \notin$ `FDOM` s $\wedge$ t $=$ `Var` v $\vee$
    $v \in$ `FDOM` s $\wedge$ $t \neq$ `Var` v $\wedge$
    $\exists u.$ `vwalk` s v $=$ `vwalk` s u $\wedge$ s `'` u $=$ `SOME` t

97. `allvars_def`

`allvars` s t_1 t_2 $=$ `vars` t_1 $\cup$ `vars` t_2 $\cup$ `substvars` s

98. `FINITE_allvars`

$\vdash$ `FINITE (allvars` s t_1 t_2`)`

99. `allvars_sym`

$\vdash$ `allvars` s t_1 t_2 $=$ `allvars` s t_2 t_1

100. `uR_def` (Definition 21)

`unify`$_{\texttt{TR}}$ `(`sx`,`c_1`,`c_2`)` `(`s`,`t_1`,`t_2`)` $\iff$
  `wfs` sx $\wedge$ $s \sqsubseteq sx$ $\wedge$ `allvars` sx c_1 c_2 $\subseteq$ `allvars` s t_1 t_2 $\wedge$
  `measure (pair_count` $\circ$ `walk*` sx`)` c_1 t_1

101. `WF_uR` (Theorem 2)

$\vdash$ `WF unify`$_{\texttt{TR}}$

102. `tunify_def` (see Figure 3.1 on page 38)

103. `vwalk_to_Pair_SUBSET_rangevars`

$\vdash$ `wfs` s $\wedge$ `vwalk` s v $=$ $\langle t_1, t_2\rangle$ $\Rightarrow$
  `vars` t_1 $\subseteq$ `rangevars` s $\wedge$ `vars` t_2 $\subseteq$ `rangevars` s

104. `allvars_SUBSET`

$\vdash$ `wfs` s $\wedge$ `walk` s t_1 $=$ $\langle t_{11}, t_{12}\rangle$ $\wedge$ `walk` s t_2 $=$ $\langle t_{21}, t_{22}\rangle$ $\Rightarrow$
  `allvars` s t_{11} t_{21} $\subseteq$ `allvars` s t_1 t_2 $\wedge$
  `allvars` s t_{12} t_{22} $\subseteq$ `allvars` s t_1 t_2

105. `walkstar_subterm_smaller`

$\vdash$ `wfs` s $\wedge$ `walk` s t_1 $=$ $\langle t_{11}, t_{12}\rangle$ $\Rightarrow$
  `measure (pair_count` $\circ$ `walk*` s`)` t_{11} t_1 $\wedge$
  `measure (pair_count` $\circ$ `walk*` s`)` t_{12} t_1

106. `tca_thm` (Lemma 19)

$\vdash$ `wfs` s $\wedge$ `walk` s t_1 $=$ $\langle t_{11}, t_{12}\rangle$ $\wedge$ `walk` s t_2 $=$ $\langle t_{21}, t_{22}\rangle$ $\Rightarrow$
  `unify`$_{\texttt{TR}}$ `(`s`,`t_{11}`,`t_{21}`)` `(`s`,`t_1`,`t_2`)`

107. `allvars_SUBSET_FUPDATE`

$\vdash$ `walk` s t_1 = $\langle t_{11}, t_{12}\rangle$ $\wedge$ `walk` s t_2 = $\langle t_{21}, t_{22}\rangle$ $\wedge$ `walk` s t_{11} = `Var` v $\wedge$
`wfs` s $\Rightarrow$
`allvars` (s `|+` (v,`walk` s t_{21})) t_{12} t_{22} $\subseteq$ `allvars` s t_1 t_2

108. `walkstar_subterm_FUPDATE`

$\vdash$ `wfs` s $\wedge$ $v \notin$ `FDOM` s $\wedge$ $v \notin$ `vars` (s $\triangleleft$ t) $\wedge$ `walk` s t_1 = $\langle t_{11}, t_{12}\rangle$ $\Rightarrow$
`measure` (`pair_count` $\circ$ `walk*` (s `|+` (v,t))) t_{12} t_1

109. `uR_subterm`

$\vdash$ `wfs` s $\wedge$ `walk` s t_1 = $\langle t_{11}, t_{12}\rangle$ $\wedge$ `walk` s t_2 = $\langle t_{21}, t_{22}\rangle$ $\Rightarrow$
$\texttt{unify}_{\texttt{TR}}$ (s,t_{11},t_{21}) (s,t_1,t_2) $\wedge$ $\texttt{unify}_{\texttt{TR}}$ (s,t_{12},t_{22}) (s,t_1,t_2)

110. `uR_subterm_FUPDATE`

$\vdash$ `walk` s t_1 = $\langle t_{11}, t_{12}\rangle$ $\wedge$ `walk` s t_2 = $\langle t_{21}, t_{22}\rangle$ $\wedge$
(`walk` s t_{11} = `Var` v $\wedge$ `walk` s t_{21} = t $\vee$
`walk` s t_{21} = `Var` v $\wedge$ `walk` s t_{11} = t) $\wedge$ `wfs` s $\wedge$ $v \notin$ `FDOM` s $\wedge$
$v \notin$ `vars` (s $\triangleleft$ t) $\Rightarrow$
$\texttt{unify}_{\texttt{TR}}$ (s `|+` (v,t),t_{12},t_{22}) (s,t_1,t_2)

111. `uP_def` (Definition 22)

$\texttt{subst}_{\texttt{R}}$ sx s t_1 t_2 $\iff$
`wfs` sx $\wedge$ $s \sqsubseteq sx$ $\wedge$ `substvars` sx $\subseteq$ `allvars` s t_1 t_2

112. `uP_FUPDATE`

$\vdash$ `wfs` s $\wedge$ $v \notin$ `FDOM` s $\wedge$ $v \notin$ `vars` (s $\triangleleft$ t_2) $\wedge$ `walk` s t_1 = `Var` v $\Rightarrow$
$\texttt{subst}_{\texttt{R}}$ (s `|+` (v,`walk` s t_2)) s t_1 t_2

113. `walk_SUBMAP`

$\vdash$ `wfs` sx $\wedge$ $s \sqsubseteq sx$ $\Rightarrow$
`case walk` s t `of`
`Var` u $\to$ `walk` sx t = `walk` sx (`Var` u)
$\|$ $\langle v_5, v_6\rangle$ $\to$ `walk` sx t = $\langle v_5, v_6\rangle$
$\|$ `Const` v_7 $\to$ `walk` sx t = `Const` v_7

114. `uR_IMP_uP`

$\vdash$ $\texttt{unify}_{\texttt{TR}}$ (sx,c_1,c_2) (s,t_1,t_2) $\Rightarrow$ $\texttt{subst}_{\texttt{R}}$ sx s t_1 t_2

115. `uP_IMP_subterm_uR` (Lemma 20)

$\vdash$ `wfs` s $\wedge$ `walk` s t_1 = $\langle t_{11}, t_{12}\rangle$ $\wedge$ `walk` s t_2 = $\langle t_{21}, t_{22}\rangle$ $\wedge$
($\texttt{subst}_{\texttt{R}}$ sx s t_{11} t_{21} $\vee$ $\texttt{subst}_{\texttt{R}}$ sx s t_{12} t_{22}) $\Rightarrow$
$\texttt{unify}_{\texttt{TR}}$ (sx,t_{12},t_{22}) (s,t_1,t_2)

116. aux_uP (Lemma 21)

```
⊢ wfs s ∧ tunify_tupled_aux unify_TR (s,t1,t2) = SOME sx ⇒
    subst_R sx s t1 t2
```

117. tcd_thm (Lemma 22)

```
⊢ wfs s ∧ walk s t1 = ⟨t11,t12⟩ ∧ walk s t2 = ⟨t21,t22⟩ ∧
  tunify_tupled_aux unify_TR (s,t11,t21) = SOME sx ⇒
    unify_TR (sx,t12,t22) (s,t1,t2)
```

118. aux_eq_tunify

```
⊢ tunify_tupled_aux unify_TR (s,t1,t2) = tunify s t1 t2
```

119. ext_s_check_def

```
ext_s_check s v t = if oc s t v then NONE else SOME (s |+ (v,t))
```

120. unify_def

```
wfs s ⇒
  unify s t1 t2 =
    case (walk s t1,walk s t2) of
      (Var v1,Var v2) →
        SOME (if v1 = v2 then s else s |+ (v1,Var v2))
    || (Var v1,⟨v27,v28⟩) → ext_s_check s v1 ⟨v27,v28⟩
    || (Var v1,Const v29) → ext_s_check s v1 (Const v29)
    || (⟨t11,t12⟩,Var v34) → ext_s_check s v34 ⟨t11,t12⟩
    || (⟨t11,t12⟩,⟨t21,t22⟩) → do sx ← unify s t11 t21; unify sx t12 t22 od
    || (⟨t11,t12⟩,Const v37) → NONE
    || (Const c1,Var v42) → ext_s_check s v42 (Const c1)
    || (Const c1,⟨v43,v44⟩) → NONE
    || (Const c1,Const c2) → if c1 = c2 then SOME s else NONE
```

121. unify_uP

```
⊢ wfs s ∧ unify s t1 t2 = SOME sx ⇒ subst_R sx s t1 t2
```

A.7 unifProps

(Omitted.)

A.8 nterm

(Omitted.)

A.9 nsubst

(Omitted.)

A.10 dis_set

(Omitted.)

A.11 apply_pi

122. `apply_pi_def` (Definition 25)

$\pi \bullet \mathtt{Nom}\ a\ =\ \mathtt{Nom}\ (\pi \bullet a)$
$\pi \bullet \mathtt{Sus}\ p\ v\ =\ \mathtt{Sus}\ (\pi\ \mathtt{++}\ p)\ v$
$\pi \bullet \mathtt{Tie}\ a\ t\ =\ \mathtt{Tie}\ (\pi \bullet a)\ (\pi \bullet t)$
$\pi \bullet \langle t_1, t_2 \rangle_{\mathtt{n}}\ =\ \langle \pi \bullet t_1, \pi \bullet t_2 \rangle_{\mathtt{n}}$
$\pi \bullet \mathtt{Const}_{\mathtt{n}}\ c\ =\ \mathtt{Const}_{\mathtt{n}}\ c$

123. `apply_pi_is_perm`

$\vdash$ `is_perm apply_pi`

124. `apply_pi_nil`

$\vdash\ \mathtt{[]} \bullet x\ =\ x$

125. `apply_pi_decompose`

$\vdash\ (x\ \mathtt{++}\ y) \bullet a\ =\ x \bullet y \bullet a$

126. `apply_pi_inverse`

$\vdash\ p \bullet p^{-1} \bullet a\ =\ a\ \wedge\ p^{-1} \bullet p \bullet a\ =\ a$

127. `apply_pi_id`

$\vdash\ ((x,x)\mathtt{::}t) \bullet a\ =\ t \bullet a$

128. `apply_pi_injective`

$\vdash\ p \bullet x\ =\ p \bullet y\ \iff\ x\ =\ y$

129. `apply_pi_eq_perms`

$\vdash\ p_1\ \mathtt{==}\ p_2\ \Rightarrow\ p_1 \bullet x\ =\ p_2 \bullet x$

130. `apply_pi_eql`

$\vdash\ p \bullet x\ =\ y\ \iff\ x\ =\ p^{-1} \bullet y$

131. `apply_pi_eqr`

$\vdash\ t_1\ =\ \pi \bullet t_2\ \iff\ \pi^{-1} \bullet t_1\ =\ t_2$

132. `nvars_apply_pi`

$\vdash\ \mathtt{nvars}\ (\pi \bullet t)\ =\ \mathtt{nvars}\ t$

A.12 nwalk

(Omitted.)

A.13 nwalkstar

(Omitted.)

A.14 nunifDef

(Omitted.)

A.15 nunifProps

133. `fresh_def` (see Definition 28 on page 51)

134. `fresh_apply_pi`

$\vdash fcs \vdash a \ \# \ t \Rightarrow fcs \vdash \pi \bullet a \ \# \ \pi \bullet t$

135. `lemma27` (see Lemma 32 on page 53)

136. `equiv_rules` (see Definition 29 on page 51)

137. `equiv_refl` (Lemma 28)

$\vdash fcs \vdash t \approx t$

138. `equiv_fresh`

$\vdash fe \vdash t_1 \approx t_2 \wedge fe \vdash a \ \# \ t_1 \Rightarrow fe \vdash a \ \# \ t_2$

139. `equiv_apply_pi`

$\vdash fe \vdash t_1 \approx t_2 \Rightarrow fe \vdash \pi \bullet t_1 \approx \pi \bullet t_2$

140. `equiv_sym` (Lemma 28)

$\vdash fe \vdash t_1 \approx t_2 \Rightarrow fe \vdash t_2 \approx t_1$

141. `fresh_ds_equiv` (Lemma 33)

$\vdash (\forall a.\ a \in \texttt{dis_set}\ \pi_1\ \pi_2 \Rightarrow fcs \vdash a \ \# \ t) \Rightarrow fcs \vdash \pi_1 \bullet t \approx \pi_2 \bullet t$

142. `equiv_ds_fresh`

$\vdash fe \vdash \pi_1 \bullet t \approx \pi_2 \bullet t \wedge a \in \texttt{dis_set}\ \pi_1\ \pi_2 \Rightarrow fe \vdash a \ \# \ t$

143. `equiv_trans_lemma` (see proof of Lemma 28)

144. `equiv_trans` (see Lemma 28) on page Lemma 28

145. `fresh_extra_fcs`

$\vdash$ *fe* $\vdash$ *a* # *t* $\wedge$ *fe* $\subseteq$ *fex* $\Rightarrow$ *fex* $\vdash$ *a* # *t*

146. `term_fcs_FINITE`

$\vdash$ `term_fcs` *a* *t* = `SOME` *fe* $\Rightarrow$ `FINITE` *fe*

147. `term_fcs_fresh` (see Lemma 29 on page 52)

148. `term_fcs_minimal` (see Lemma 30)

149. `term_fcs_NONE` (see Lemma 31)

150. `fcs_acc_def` (from nunifDef)

```
fcs_acc s (a,v) ac =
  OPTION_MAP2 (∪) ac (term_fcs a (s ◁n Sus [] v))
```

151. `fcs_acc_RECURSES`

```
⊢ FINITE t ⇒
    ITSET (fcs_acc s) (e INSERT t) b =
      fcs_acc s e (ITSET (fcs_acc s) (t DELETE e) b)
```

152. `unify_eq_vars_extends_fe`

```
⊢ FINITE ds ∧ unify_eq_vars ds v (s,fe) = SOME (s′,fex) ⇒ fe ⊆ fex
```

153. `unify_eq_vars_FINITE`

```
⊢ FINITE ds ∧ unify_eq_vars ds v (s,fe) = SOME (s′,fex) ∧
  FINITE fe ⇒
    FINITE fex
```

154. `unify_eq_vars_fresh`

```
⊢ FINITE ds ∧ unify_eq_vars ds v (s,fe) = SOME (s′,fex) ∧ a ∈ ds ⇒
    fex ⊢ a # s′ ◁n Sus [] v
```

155. `unify_eq_vars_minimal`

```
⊢ FINITE ds ∧ unify_eq_vars ds v (s,fe) = SOME (s′,fex) ∧ c ∈ fex ⇒
    c ∈ fe ∨
    c ∉ fe ∧
    ∃a fcs. term_fcs a (s ◁n Sus [] v) = SOME fcs ∧ a ∈ ds ∧ c ∈ fcs
```

156. `unify_eq_vars_NONE`

```
⊢ FINITE ds ∧ unify_eq_vars ds v (s,fe) = NONE ⇒
    ∃a. a ∈ ds ∧ ∀fe. fe ⊬ a # s ◁n Sus [] v
```

157. `unify_eq_vars_decompose`

$$\begin{aligned}&\vdash \mathtt{unify_eq_vars}\ (a\ \mathtt{INSERT}\ ds)\ v\ (s,fe) = \mathtt{SOME}\ (sx,fex) \land{}\\ &\quad \mathtt{FINITE}\ ds \Rightarrow\\ &\qquad \exists fe_0\ fcs.\\ &\qquad\quad \mathtt{unify_eq_vars}\ ds\ v\ (s,fe) = \mathtt{SOME}\ (s,fe_0) \land{}\\ &\qquad\quad \mathtt{term_fcs}\ a\ (s \lhd_{\mathrm{n}} \mathtt{Sus}\ \mathtt{[]}\ v) = \mathtt{SOME}\ fcs \land fex = fe_0 \cup fcs\end{aligned}$$

158. `unify_eq_vars_INSERT`

$$\begin{aligned}&\vdash \mathtt{unify_eq_vars}\ ds\ v\ (s,fe) = \mathtt{SOME}\ (sx,fex) \land{}\\ &\quad \mathtt{term_fcs}\ a\ (s \lhd_{\mathrm{n}} \mathtt{Sus}\ \mathtt{[]}\ v) = \mathtt{SOME}\ fcs \land \mathtt{FINITE}\ ds \Rightarrow\\ &\qquad \mathtt{unify_eq_vars}\ (a\ \mathtt{INSERT}\ ds)\ v\ (s,fe) = \mathtt{SOME}\ (s,fex \cup fcs)\end{aligned}$$

159. `equiv_eq_perms`

$$\begin{aligned}&\vdash fcs \vdash \mathtt{Sus}\ p_1\ v_1 \approx \mathtt{Sus}\ p_2\ v_2 \land p_1\ \mathtt{==}\ q_1 \land p_2\ \mathtt{==}\ q_2 \Rightarrow\\ &\qquad fcs \vdash \mathtt{Sus}\ q_1\ v_1 \approx \mathtt{Sus}\ q_2\ v_2\end{aligned}$$

160. `fresh_eq_perms`

$$\vdash fcs \vdash a\ \#\ \mathtt{Sus}\ p\ v \land p\ \mathtt{==}\ q \Rightarrow fcs \vdash a\ \#\ \mathtt{Sus}\ q\ v$$

161. `unify_eq_vars_equiv` (see Lemma 34 on page 55)

162. `nunify_extends_fe`

$$\vdash \mathtt{wfs}_{\mathrm{n}}\ s \land \mathtt{unify}_{\mathrm{n}}\ (s,fe)\ t_1\ t_2 = \mathtt{SOME}\ (sx,fex) \Rightarrow fe \subseteq fex$$

163. `unify_eq_vars_empty`

$$\vdash \mathtt{unify_eq_vars}\ \{\}\ v\ (s,fe) = \mathtt{SOME}\ (s,fe)$$

164. `verify_fcs_empty`

$$\vdash \mathtt{verify_fcs}\ \{\}\ s = \mathtt{SOME}\ \{\}$$

165. `verify_fcs_FINITE`

$$\vdash \mathtt{FINITE}\ fe \land \mathtt{verify_fcs}\ fe\ s = \mathtt{SOME}\ ve \Rightarrow \mathtt{FINITE}\ ve$$

166. `verify_fcs_covers_all`

$$\begin{aligned}&\vdash \mathtt{FINITE}\ fe \land \mathtt{verify_fcs}\ fe\ s = \mathtt{SOME}\ ve \land (a,v) \in fe \Rightarrow\\ &\qquad \exists fcs.\ \mathtt{term_fcs}\ a\ (s \lhd_{\mathrm{n}} \mathtt{Sus}\ \mathtt{[]}\ v) = \mathtt{SOME}\ fcs \land fcs \subseteq ve\end{aligned}$$

167. `verify_fcs_minimal`

$$\begin{aligned}&\vdash \mathtt{FINITE}\ fe \land \mathtt{verify_fcs}\ fe\ s = \mathtt{SOME}\ ve \land (a,v) \in ve \Rightarrow\\ &\qquad \exists b\ w\ fcs.\\ &\qquad\quad (b,w) \in fe \land \mathtt{term_fcs}\ b\ (s \lhd_{\mathrm{n}} \mathtt{Sus}\ \mathtt{[]}\ w) = \mathtt{SOME}\ fcs \land{}\\ &\qquad\quad (a,v) \in fcs\end{aligned}$$

168. `verify_fcs_NONE`

$\vdash$ FINITE fe $\wedge$ verify_fcs fe s = NONE $\Rightarrow$
  $\exists\, a\ v.$ $(a,v) \in fe$ $\wedge$ term_fcs a $(s \lhd_{\mathtt{n}}$ Sus [] $v)$ = NONE

169. `verify_fcs_INSERT`

$\vdash$ FINITE fe $\wedge$ verify_fcs fe s = SOME ve $\wedge$
 term_fcs a $(s \lhd_{\mathtt{n}}$ Sus [] $v)$ = SOME fcs $\Rightarrow$
  verify_fcs $((a,v)$ INSERT $fe)$ s = SOME $(ve \cup fcs)$

170. `verify_fcs_decompose`

$\vdash$ verify_fcs $((a,v)$ INSERT $fe)$ s = SOME ve $\wedge$ FINITE fe $\Rightarrow$
  $\exists\, ve_0\ fcs.$
   verify_fcs fe s = SOME ve_0 $\wedge$
   term_fcs a $(s \lhd_{\mathtt{n}}$ Sus [] $v)$ = SOME fcs $\wedge$ $ve = ve_0 \cup fcs$

171. `verify_fcs_UNION_I`

$\vdash$ FINITE fe_1 $\wedge$ FINITE fe_2 $\wedge$ verify_fcs fe_1 s = SOME ve_1 $\wedge$
 verify_fcs fe_2 s = SOME ve_2 $\Rightarrow$
  verify_fcs $(fe_1 \cup fe_2)$ s = SOME $(ve_1 \cup ve_2)$

172. `verify_fcs_UNION_E`

$\vdash$ FINITE $(fe_1 \cup fe_2)$ $\wedge$ verify_fcs $(fe_1 \cup fe_2)$ s = SOME ve $\Rightarrow$
  $\exists\, ve_1\ ve_2.$
   verify_fcs fe_1 s = SOME ve_1 $\wedge$ verify_fcs fe_2 s = SOME ve_2 $\wedge$
   $ve = ve_1 \cup ve_2$

173. `verify_fcs_UNION`

$\vdash$ FINITE fe_1 $\wedge$ FINITE fe_2 $\Rightarrow$
  $((\exists\, ve_1\ ve_2.$
    verify_fcs fe_1 s = SOME ve_1 $\wedge$ verify_fcs fe_2 s = SOME ve_2 $\wedge$
    $ve = ve_1 \cup ve_2)$ $\iff$
   verify_fcs $(fe_1 \cup fe_2)$ s = SOME $ve)$

174. `fresh_verify_fcs` (Lemma 35)

$\vdash$ wfs$_{\mathtt{n}}$ s $\wedge$ $fe \vdash a \# t$ $\wedge$ FINITE fe $\wedge$ verify_fcs fe s = SOME fex $\Rightarrow$
  $fex \vdash a \# s \lhd_{\mathtt{n}} t$

175. `nwalkstar_subterm_exists`

$\vdash$ wfs$_{\mathtt{n}}$ s $\Rightarrow$
  $\forall\, t.$
   $(\forall\, a\ c.\ s \lhd_{\mathtt{n}} t =$ Tie $a\ c \Rightarrow \exists\, t_2.\ s \lhd_{\mathtt{n}} t_2 = c)$ $\wedge$
   $\forall\, c_1\ c_2.$
    $s \lhd_{\mathtt{n}} t = \langle c_1, c_2 \rangle_{\mathtt{n}} \Rightarrow (\exists\, t_2.\ s \lhd_{\mathtt{n}} t_2 = c_1) \wedge \exists\, t_2.\ s \lhd_{\mathtt{n}} t_2 = c_2$

176. `equiv_verify_fcs` (Lemma 36)

$\vdash fe \vdash t_1 \approx t_2 \wedge \texttt{wfs}_\texttt{n}\ s \wedge \texttt{FINITE}\ fe \wedge \texttt{verify_fcs}\ fe\ s = \texttt{SOME}\ fex \Rightarrow$
$\quad fex \vdash s \lhd_\texttt{n} t_1 \approx s \lhd_\texttt{n} t_2$

177. `nunify_FINITE_fe`

$\vdash \texttt{wfs}_\texttt{n}\ s \wedge \texttt{FINITE}\ fe \wedge \texttt{unify}_\texttt{n}\ (s,fe)\ t_1\ t_2 = \texttt{SOME}\ (sx,fex) \Rightarrow$
$\quad \texttt{FINITE}\ fex$

178. `fresh_SUBMAP`

$\vdash fcs \vdash a \# s \lhd_\texttt{n} t \wedge \texttt{wfs}_\texttt{n}\ s \Rightarrow$
$\quad \exists fe.\ fe \vdash a \# t \wedge \forall b\ w.\ (b,w) \in fe \Rightarrow fcs \vdash b \# s \lhd_\texttt{n} \texttt{Sus []}\ w$

179. `equiv_SUBSET`

$\vdash fe \vdash t_1 \approx t_2 \wedge fe \subseteq fex \Rightarrow fex \vdash t_1 \approx t_2$

180. `fresh_drop_NOTIN_nvars`

$\vdash fe \vdash a \# t \wedge v \notin \texttt{nvars}\ t \Rightarrow fe\ \texttt{DIFF}\ \{(b,u) \mid u = v\} \vdash a \# t$

181. `term_fcs_IN_nvars`

$\vdash \texttt{term_fcs}\ a\ t = \texttt{SOME}\ fe \wedge (b,v) \in fe \Rightarrow v \in \texttt{nvars}\ t$

182. `fresh_drop_IN_FDOM`

$\vdash \texttt{wfs}_\texttt{n}\ s \wedge fe \vdash a \# s \lhd_\texttt{n} t \wedge v \in \texttt{FDOM}\ s \Rightarrow$
$\quad fe\ \texttt{DIFF}\ \{(b,u) \mid u = v\} \vdash a \# s \lhd_\texttt{n} t$

183. `term_fcs_NOTIN_FDOM`

$\vdash \texttt{wfs}_\texttt{n}\ s \wedge \texttt{term_fcs}\ a\ (s \lhd_\texttt{n} t) = \texttt{SOME}\ fe \wedge (b,v) \in fe \Rightarrow v \notin \texttt{FDOM}\ s$

184. `verify_fcs_NOTIN_FDOM`

$\vdash \texttt{verify_fcs}\ fe\ s = \texttt{SOME}\ ve \wedge (a,v) \in ve \wedge \texttt{wfs}_\texttt{n}\ s \wedge \texttt{FINITE}\ fe \Rightarrow$
$\quad v \notin \texttt{FDOM}\ s$

185. `verify_fcs_SUBSET`

$\vdash \texttt{FINITE}\ fex \wedge \texttt{verify_fcs}\ fex\ s = \texttt{SOME}\ vex \wedge fe \subseteq fex \Rightarrow$
$\quad \exists ve.\ \texttt{verify_fcs}\ fe\ s = \texttt{SOME}\ ve \wedge ve \subseteq vex$

186. `term_fcs_apply_pi`

$\vdash \texttt{term_fcs}\ a\ t = \texttt{SOME}\ fcs \Rightarrow \texttt{term_fcs}\ (\pi \bullet a)\ (\pi \bullet t) = \texttt{SOME}\ fcs$

187. `term_fcs_nwalkstar`

$\vdash \texttt{wfs}_\texttt{n}\ s \wedge \texttt{term_fcs}\ b\ t = \texttt{SOME}\ fcs \wedge$
$\quad \texttt{term_fcs}\ b\ (s \lhd_\texttt{n} t) = \texttt{SOME}\ fcs_2 \wedge (a,v) \in fcs \Rightarrow$
$\quad\quad \exists fcs_1.\ \texttt{term_fcs}\ a\ (s \lhd_\texttt{n} \texttt{Sus []}\ v) = \texttt{SOME}\ fcs_1 \wedge fcs_1 \subseteq fcs_2$

188. `verify_fcs_SUBMAP`

$$\begin{array}{l}\vdash \mathtt{FINITE}\ fe \land \mathtt{verify_fcs}\ fe\ sx = \mathtt{SOME}\ fex \land \mathtt{wfs_n}\ sx \land s \sqsubseteq sx \Rightarrow \\ \quad \exists fe_1. \\ \quad\quad \mathtt{verify_fcs}\ fe\ s = \mathtt{SOME}\ fe_1 \land \\ \quad\quad \forall a\ v. \\ \quad\quad\quad (a,v) \in fe_1 \Rightarrow \\ \quad\quad\quad\quad \exists fcs.\ \mathtt{term_fcs}\ a\ (sx \lhd_{\mathtt{n}} \mathtt{Sus}\ []\ v) = \mathtt{SOME}\ fcs \land fcs \subseteq fex\end{array}$$

189. `verify_fcs_term_fcs`

$$\begin{array}{l}\vdash \mathtt{term_fcs}\ a\ t = \mathtt{SOME}\ fcs \land \mathtt{term_fcs}\ a\ (s \lhd_{\mathtt{n}} t) = \mathtt{SOME}\ fex \land \mathtt{wfs_n}\ s \Rightarrow \\ \quad \mathtt{verify_fcs}\ fcs\ s = \mathtt{SOME}\ fex\end{array}$$

190. `verify_fcs_iter_SUBMAP` (Lemma 37)

$$\begin{array}{l}\vdash \mathtt{verify_fcs}\ fe\ s = \mathtt{SOME}\ ve_0 \land \mathtt{verify_fcs}\ fe\ sx = \mathtt{SOME}\ ve \land \\ \quad s \sqsubseteq sx \land \mathtt{wfs_n}\ sx \land \mathtt{FINITE}\ fe \Rightarrow \\ \quad\quad \mathtt{verify_fcs}\ ve_0\ sx = \mathtt{SOME}\ ve\end{array}$$

191. `verify_fcs_idem`

$$\begin{array}{l}\vdash \mathtt{wfs_n}\ s \land \mathtt{FINITE}\ fe \land \mathtt{verify_fcs}\ fe\ s = \mathtt{SOME}\ fex \Rightarrow \\ \quad \mathtt{verify_fcs}\ fex\ s = \mathtt{SOME}\ fex\end{array}$$

192. `unify_eq_vars_adds_same_fcs`

$$\begin{array}{l}\vdash \mathtt{FINITE}\ ds \Rightarrow \\ \quad \exists fu. \\ \quad\quad \forall fe\ sx\ fex. \\ \quad\quad\quad \mathtt{unify_eq_vars}\ ds\ v\ (s,fe) = \mathtt{SOME}\ (sx,fex) \Rightarrow fex = fe \cup fu\end{array}$$

193. `unify_eq_vars_ignores_fe`

$$\begin{array}{l}\vdash \mathtt{unify_eq_vars}\ ds\ v\ (s,fe) = \mathtt{SOME}\ (sx,fex) \land \mathtt{FINITE}\ ds \Rightarrow \\ \quad \forall fe'.\ \exists fex'.\ \mathtt{unify_eq_vars}\ ds\ v\ (s,fe') = \mathtt{SOME}\ (s,fex')\end{array}$$

194. `nunify_ignores_fe`

$$\begin{array}{l}\vdash \mathtt{wfs_n}\ s \land \mathtt{unify_n}\ (s,fe)\ t_1\ t_2 = \mathtt{SOME}\ (sx,fex) \Rightarrow \\ \quad \forall fe'.\ \exists fex'.\ \mathtt{unify_n}\ (s,fe')\ t_1\ t_2 = \mathtt{SOME}\ (sx,fex')\end{array}$$

195. `nunify_ignores_fe_NONE`

$$\begin{array}{l}\vdash \mathtt{wfs_n}\ s \land \mathtt{unify_n}\ (s,fe)\ t_1\ t_2 = \mathtt{NONE} \Rightarrow \\ \quad \forall fe'.\ \mathtt{unify_n}\ (s,fe')\ t_1\ t_2 = \mathtt{NONE}\end{array}$$

196. `nunify_adds_same_fcs`

$$\begin{array}{l}\vdash \mathtt{wfs_n}\ s \Rightarrow \\ \quad \exists fu. \\ \quad\quad \forall fe\ sx\ fex. \\ \quad\quad\quad \mathtt{unify_n}\ (s,fe)\ t_1\ t_2 = \mathtt{SOME}\ (sx,fex) \Rightarrow fex = fe \cup fu\end{array}$$

197. `nomunify_unifier` (Theorem 6)

$\vdash \mathtt{wfs_n}\ s \land \mathtt{FINITE}\ \mathit{fe} \land \mathtt{nomunify}\ (s,\mathit{fe})\ t_1\ t_2 = \mathtt{SOME}\ (\mathit{sx},\mathit{fex}) \Rightarrow$
$\quad \mathtt{FINITE}\ \mathit{fex} \land \mathtt{wfs_n}\ \mathit{sx} \land s \sqsubseteq \mathit{sx} \land \mathit{fex} \vdash \mathit{sx} \lhd_\mathtt{n} t_1 \approx \mathit{sx} \lhd_\mathtt{n} t_2$

198. `nomunify_fcs_consistent`

$\vdash \mathtt{wfs_n}\ s \land \mathtt{FINITE}\ \mathit{fe} \land \mathtt{nomunify}\ (s,\mathit{fe})\ t_1\ t_2 = \mathtt{SOME}\ (\mathit{sx},\mathit{fex}) \land$
$(a,v) \in \mathit{fex} \Rightarrow$
$\quad \mathit{fex} \vdash a \mathrel{\#} \mathit{sx} \lhd_\mathtt{n} \mathtt{Sus}\ []\ v$

199. `nomunify_solves_fe`

$\vdash \mathtt{wfs_n}\ s \land \mathtt{FINITE}\ \mathit{fe} \land \mathtt{nomunify}\ (s,\mathit{fe})\ t_1\ t_2 = \mathtt{SOME}\ (\mathit{sx},\mathit{fex}) \land$
$(a,v) \in \mathit{fe} \Rightarrow$
$\quad \mathit{fex} \vdash a \mathrel{\#} \mathit{sx} \lhd_\mathtt{n} \mathtt{Sus}\ []\ v$

200. `nvwalk_irrelevant_FUPDATE`

$\vdash \mathtt{wfs_n}\ (s \mathbin{|+} (\mathit{vx},\mathit{tx})) \land \mathit{vx} \notin \mathtt{FDOM}\ s \Rightarrow$
$\quad \forall p\ v.\ \neg(\mathtt{nvR}\ s)^*\ \mathit{vx}\ v \Rightarrow \mathtt{vwalk_n}\ (s \mathbin{|+} (\mathit{vx},\mathit{tx}))\ p\ v = \mathtt{vwalk_n}\ s\ p\ v$

201. `TC_nvR_SUBMAP`

$\vdash s \sqsubseteq \mathit{sx} \Rightarrow \forall u\ v.\ (\mathtt{nvR}\ s)^+\ u\ v \Rightarrow (\mathtt{nvR}\ \mathit{sx})^+\ u\ v$

202. `nvwalk_FUPDATE_var`

$\vdash \mathtt{wfs_n}\ (s \mathbin{|+} (\mathit{vx},\mathit{tx})) \land \mathit{vx} \notin \mathtt{FDOM}\ s \Rightarrow$
$\quad \mathtt{vwalk_n}\ (s \mathbin{|+} (\mathit{vx},\mathit{tx}))\ []\ \mathit{vx} = \mathtt{walk_n}\ s\ \mathit{tx}$

203. `nwalkstar_irrelevant_FUPDATE`

$\vdash \mathtt{wfs_n}\ (s \mathbin{|+} (\mathit{vx},\mathit{tx})) \land \mathit{vx} \notin \mathtt{FDOM}\ s \land \neg\mathtt{oc_n}\ s\ t\ \mathit{vx} \Rightarrow$
$\quad (s \mathbin{|+} (\mathit{vx},\mathit{tx})) \lhd_\mathtt{n} t = s \lhd_\mathtt{n} t$

204. `extension_is_nwfs`

$\vdash \mathtt{wfs_n}\ (s \mathbin{|+} (\mathit{vx},\mathit{tx})) \land \mathit{vx} \notin \mathtt{FDOM}\ s \Rightarrow \mathtt{wfs_n}\ s \land \neg\mathtt{oc_n}\ s\ \mathit{tx}\ \mathit{vx}$

205. `nwalkstar_FUPDATE_var`

$\vdash \mathtt{wfs_n}\ (s \mathbin{|+} (\mathit{vx},\mathit{tx})) \land \mathit{vx} \notin \mathtt{FDOM}\ s \Rightarrow$
$\quad (s \mathbin{|+} (\mathit{vx},\mathit{tx})) \lhd_\mathtt{n} \mathtt{Sus}\ []\ \mathit{vx} = s \lhd_\mathtt{n} \mathit{tx}$

206. `equiv_extend`

$\vdash \mathtt{wfs_n}\ s_1 \land \mathtt{wfs_n}\ (s \mathbin{|+} (\mathit{vx},\pi^{-1} \bullet \mathit{tx})) \land \mathit{vx} \notin \mathtt{FDOM}\ s \land$
$\mathit{fe} \vdash s_1 \lhd_\mathtt{n} \mathtt{Sus}\ \pi\ \mathit{vx} \approx s_1 \lhd_\mathtt{n} s \lhd_\mathtt{n} \mathit{tx} \Rightarrow$
$\quad \forall t.\ \mathit{fe} \vdash s_1 \lhd_\mathtt{n} (s \mathbin{|+} (\mathit{vx},\pi^{-1} \bullet \mathit{tx})) \lhd_\mathtt{n} t \approx s_1 \lhd_\mathtt{n} s \lhd_\mathtt{n} t$

207. `nvars_nwalkstar_subterm`

$\vdash \texttt{wfs}_{\texttt{n}}\ s \Rightarrow$
$((\exists v_1.\ v_1 \in \texttt{nvars}\ t \land v_2 \in \texttt{nvars}\ (s \lhd_{\texttt{n}} \texttt{Sus []}\ v_1)) \iff$
$v_2 \in \texttt{nvars}\ (s \lhd_{\texttt{n}} t))$

208. `equiv_ties_fcs`

$\vdash fe \vdash \texttt{Tie}\ a_1\ t_1 \approx \texttt{Tie}\ a_2\ t_2 \land a_1 \neq a_2 \Rightarrow$
$\exists fcs.\ \texttt{term_fcs}\ a_1\ t_2 = \texttt{SOME}\ fcs \land fcs \subseteq fe$

209. `fresh_FINITE`

$\vdash fe \vdash a \# t \Rightarrow \exists fcs.\ fcs \subseteq fe \land \texttt{FINITE}\ fcs \land fcs \vdash a \# t$

210. `equiv_FINITE`

$\vdash fe \vdash t_1 \approx t_2 \Rightarrow \exists fcs.\ fcs \subseteq fe \land \texttt{FINITE}\ fcs \land fcs \vdash t_1 \approx t_2$

211. `term_fcs_irrelevant_nwalkstar`

$\vdash \texttt{term_fcs}\ b\ t = \texttt{SOME}\ fcs \land \texttt{term_fcs}\ b\ (s \lhd_{\texttt{n}} t) = \texttt{SOME}\ fcs_2 \land$
$(a,v) \in fcs \land v \notin \texttt{FDOM}\ s \land \texttt{wfs}_{\texttt{n}}\ s \Rightarrow$
$(a,v) \in fcs_2$

212. `nomunify_mgu2`

$\vdash \texttt{nomunify}\ (s,fe)\ t_1\ t_2 = \texttt{SOME}\ (sx,fex) \land$
$fe_2 \vdash s_2 \lhd_{\texttt{n}} s \lhd_{\texttt{n}} t_1 \approx s_2 \lhd_{\texttt{n}} s \lhd_{\texttt{n}} t_2 \land \texttt{wfs}_{\texttt{n}}\ s_2 \land \texttt{wfs}_{\texttt{n}}\ s \land \texttt{FINITE}\ fe \Rightarrow$
$\forall t.\ fe_2 \vdash s_2 \lhd_{\texttt{n}} sx \lhd_{\texttt{n}} t \approx s_2 \lhd_{\texttt{n}} s \lhd_{\texttt{n}} t$

213. `term_fcs_equiv` (Lemma 40)

$\vdash fe \vdash t_1 \approx t_2 \land \texttt{term_fcs}\ a\ t_1 = \texttt{SOME}\ fcs_1 \land fcs_1 \subseteq fe \Rightarrow$
$\exists fcs_2.\ \texttt{term_fcs}\ a\ t_2 = \texttt{SOME}\ fcs_2 \land fcs_2 \subseteq fe$

214. `nvwalk_SUBMAP_idem`

$\vdash \texttt{wfs}_{\texttt{n}}\ sx \land s \sqsubseteq sx \Rightarrow \forall \pi\ v.\ \texttt{walk}_{\texttt{n}}\ s\ (\texttt{vwalk}_{\texttt{n}}\ sx\ \pi\ v) = \texttt{vwalk}_{\texttt{n}}\ sx\ \pi\ v$

215. `nwalkstar_SUBMAP_idem`

$\vdash \texttt{wfs}_{\texttt{n}}\ sx \land s \sqsubseteq sx \Rightarrow \forall t.\ s \lhd_{\texttt{n}} sx \lhd_{\texttt{n}} t = sx \lhd_{\texttt{n}} t$

216. `unify_eq_vars_NOTIN_FDOM`

$\vdash \texttt{FINITE}\ ds \land \texttt{wfs}_{\texttt{n}}\ s \land \texttt{unify_eq_vars}\ ds\ v\ (s,fe) = \texttt{SOME}\ (s',fex) \land$
$(b,w) \notin fe \land (b,w) \in fex \Rightarrow$
$w \notin \texttt{FDOM}\ s$

217. nunify_fcs_NOTIN_FDOM

$\vdash \texttt{wfs}_{\texttt{n}}\ s \land \texttt{FINITE}\ fe \land \texttt{unify}_{\texttt{n}}\ (s,fe)\ t_1\ t_2 = \texttt{SOME}\ (sx,fex) \land$
$\quad (a,v) \notin fe \land (a,v) \in fex \Rightarrow$
$\quad\quad v \notin \texttt{FDOM}\ s$

218. fresh_term_fcs

$\vdash fe \vdash a \# t \Rightarrow \exists fcs.\ \texttt{term_fcs}\ a\ t = \texttt{SOME}\ fcs \land fcs \subseteq fe$

219. equiv_fcs_def

$\texttt{equiv_fcs}\ (\texttt{Nom}\ a)\ (\texttt{Nom}\ b) = (\text{if}\ a = b\ \text{then}\ \texttt{SOME}\ \{\,\}\ \text{else}\ \texttt{NONE}) \land$
$\texttt{equiv_fcs}\ (\texttt{Sus}\ p_1\ v_1)\ (\texttt{Sus}\ p_2\ v_2) =$
$\quad (\text{if}\ v_1 = v_2\ \text{then}\ \texttt{SOME}\ \{(a,v_1) \mid a \in \texttt{dis_set}\ p_1\ p_2\}\ \text{else}\ \texttt{NONE}) \land$
$\texttt{equiv_fcs}\ (\texttt{Tie}\ a_1\ t_1)\ (\texttt{Tie}\ a_2\ t_2) =$
$\quad (\text{if}\ a_1 = a_2\ \text{then}$
$\quad\quad \texttt{equiv_fcs}\ t_1\ t_2$
$\quad\ \text{else}$
$\quad\quad \texttt{OPTION_MAP2}\ (\cup)\ (\texttt{term_fcs}\ a_1\ t_2)$
$\quad\quad\quad (\texttt{equiv_fcs}\ t_1\ ([(a_1,a_2)] \bullet t_2))) \land$
$\texttt{equiv_fcs}\ \langle t_{11}, t_{12}\rangle_{\texttt{n}}\ \langle t_{21}, t_{22}\rangle_{\texttt{n}} =$
$\quad \texttt{OPTION_MAP2}\ (\cup)\ (\texttt{equiv_fcs}\ t_{11}\ t_{21})\ (\texttt{equiv_fcs}\ t_{12}\ t_{22}) \land$
$\texttt{equiv_fcs}\ (\texttt{Const}_{\texttt{n}}\ c_1)\ (\texttt{Const}_{\texttt{n}}\ c_2) =$
$\quad (\text{if}\ c_1 = c_2\ \text{then}\ \texttt{SOME}\ \{\,\}\ \text{else}\ \texttt{NONE}) \land \texttt{equiv_fcs}\ t_1\ t_2 = \texttt{NONE}$

220. equiv_fcs_minimal (Lemma 38)

$\vdash fe \vdash t_1 \approx t_2 \Rightarrow \exists fe_0.\ \texttt{equiv_fcs}\ t_1\ t_2 = \texttt{SOME}\ fe_0 \land fe_0 \subseteq fe$

221. equiv_fcs_equiv

$\vdash \texttt{equiv_fcs}\ t_1\ t_2 = \texttt{SOME}\ fe \Rightarrow fe \vdash t_1 \approx t_2$

222. nwalk_apply_pi

$\vdash \texttt{wfs}_{\texttt{n}}\ s \Rightarrow \texttt{walk}_{\texttt{n}}\ s\ (\pi \bullet t) = \pi \bullet \texttt{walk}_{\texttt{n}}\ s\ t$

223. equiv_fcs_nwalkstar

$\vdash \texttt{equiv_fcs}\ t_1\ t_2 = \texttt{SOME}\ fe_1 \land$
$\quad \texttt{equiv_fcs}\ (s \lhd_{\texttt{n}} t_1)\ (s \lhd_{\texttt{n}} t_2) = \texttt{SOME}\ fe_2 \land (a,v) \in fe_1 \land \texttt{wfs}_{\texttt{n}}\ s \Rightarrow$
$\quad\quad \exists fcs.\ \texttt{term_fcs}\ a\ (s \lhd_{\texttt{n}} \texttt{Sus}\ []\ v) = \texttt{SOME}\ fcs \land fcs \subseteq fe_2$

224. term_fcs_SUBMAP

$\vdash \texttt{term_fcs}\ a\ (s \lhd_{\texttt{n}} t) = \texttt{SOME}\ fcs \land \texttt{wfs}_{\texttt{n}}\ s \Rightarrow$
$\quad\quad \exists fe_0.$
$\quad\quad\quad \texttt{term_fcs}\ a\ t = \texttt{SOME}\ fe_0 \land$
$\quad\quad\quad \forall b\ w.$
$\quad\quad\quad\quad (b,w) \in fe_0 \Rightarrow$
$\quad\quad\quad\quad\quad \exists fe_1.\ \texttt{term_fcs}\ b\ (s \lhd_{\texttt{n}} \texttt{Sus}\ []\ w) = \texttt{SOME}\ fe_1 \land fe_1 \subseteq fcs$

225. `term_fcs_nwalkstar_backwards`

$$\begin{array}{l}
\vdash \texttt{wfs}_{\texttt{n}}\ s \land \texttt{term_fcs}\ b\ (s \lhd_{\texttt{n}}\ t) = \texttt{SOME}\ \mathit{fcs} \land (a,v) \in \mathit{fcs} \land \\
\quad \texttt{term_fcs}\ b\ t = \texttt{SOME}\ \mathit{fe}_0 \Rightarrow \\
\qquad \exists c\ w\ \mathit{fe}_1. \\
\qquad\quad (c,w) \in \mathit{fe}_0 \land \texttt{term_fcs}\ c\ (s \lhd_{\texttt{n}}\ \texttt{Sus []}\ w) = \texttt{SOME}\ \mathit{fe}_1 \land \\
\qquad\quad (a,v) \in \mathit{fe}_1
\end{array}$$

226. `nomunify_equiv_fcs` (Lemma 39)

$$\begin{array}{l}
\vdash \texttt{nomunify}\ (s,\mathit{fe})\ t_1\ t_2 = \texttt{SOME}\ (\mathit{sx},\mathit{fex}) \land (a,v) \in \mathit{fex} \land \texttt{wfs}_{\texttt{n}}\ s \land \\
\quad \texttt{FINITE}\ \mathit{fe} \Rightarrow \\
\qquad (\exists b\ w\ \mathit{fcs}. \\
\qquad\quad (b,w) \in \mathit{fe} \land \texttt{term_fcs}\ b\ (\mathit{sx} \lhd_{\texttt{n}}\ \texttt{Sus []}\ w) = \texttt{SOME}\ \mathit{fcs} \land \\
\qquad\quad (a,v) \in \mathit{fcs}) \lor \\
\qquad (a,v) \in \texttt{THE}\ (\texttt{equiv_fcs}\ (\mathit{sx} \lhd_{\texttt{n}}\ t_1)\ (\mathit{sx} \lhd_{\texttt{n}}\ t_2))
\end{array}$$

227. `nomunify_mgu1`

$$\begin{array}{l}
\vdash \texttt{nomunify}\ (s,\mathit{fe})\ t_1\ t_2 = \texttt{SOME}\ (\mathit{sx},\mathit{fex}) \land (a,v) \in \mathit{fex} \land \\
\quad \mathit{fe}_2 \vdash s_2 \lhd_{\texttt{n}}\ s \lhd_{\texttt{n}}\ t_1 \approx s_2 \lhd_{\texttt{n}}\ s \lhd_{\texttt{n}}\ t_2 \land \texttt{wfs}_{\texttt{n}}\ s_2 \land \texttt{wfs}_{\texttt{n}}\ s \land \texttt{FINITE}\ \mathit{fe} \Rightarrow \\
\qquad (\exists b\ w\ \mathit{fcs}. \\
\qquad\quad (b,w) \in \mathit{fe} \land \texttt{term_fcs}\ b\ (\mathit{sx} \lhd_{\texttt{n}}\ \texttt{Sus []}\ w) = \texttt{SOME}\ \mathit{fcs} \land \\
\qquad\quad (a,v) \in \mathit{fcs}) \lor \\
\qquad \mathit{fe}_2 \vdash a \mathrel{\#} s_2 \lhd_{\texttt{n}}\ \texttt{Sus []}\ v
\end{array}$$

228. `nomunify_mgu` (see Theorem 7 on 57)

229. `equiv_consistent_exists`

$$\begin{array}{l}
\vdash \mathit{fe} \vdash s \lhd_{\texttt{n}}\ t_1 \approx s \lhd_{\texttt{n}}\ t_2 \land \texttt{wfs}_{\texttt{n}}\ s \Rightarrow \\
\qquad \exists \mathit{fec}. \\
\qquad\quad \texttt{FINITE}\ \mathit{fec} \land \texttt{verify_fcs}\ \mathit{fec}\ s = \texttt{SOME}\ \mathit{fec} \land \\
\qquad\quad \mathit{fec} \vdash s \lhd_{\texttt{n}}\ t_1 \approx s \lhd_{\texttt{n}}\ t_2
\end{array}$$

230. `unify_eq_vars_verify_fcs`

$$\begin{array}{l}
\vdash \texttt{verify_fcs}\ \mathit{fe}\ s = \texttt{SOME}\ \mathit{ve} \land \texttt{wfs}_{\texttt{n}}\ s \land \texttt{FINITE}\ \mathit{ds} \land \texttt{FINITE}\ \mathit{fe} \land \\
\quad \texttt{unify_eq_vars}\ \mathit{ds}\ v\ (s,\mathit{fe}) = \texttt{SOME}\ (s',\mathit{fe}') \Rightarrow \\
\qquad \exists \mathit{ve}'.\ \texttt{verify_fcs}\ \mathit{fe}'\ s' = \texttt{SOME}\ \mathit{ve}'
\end{array}$$

231. `nwalk_nwalkstar`

$$\vdash \texttt{wfs}_{\texttt{n}}\ s \Rightarrow \forall t.\ \texttt{walk}_{\texttt{n}}\ s\ (s \lhd_{\texttt{n}}\ t) = s \lhd_{\texttt{n}}\ t$$

232. `nomunify_eqs`

$$\begin{array}{l}
\vdash \texttt{walk}_{\texttt{n}}\ s\ t_1 = q_1 \bullet t \land \texttt{walk}_{\texttt{n}}\ s\ t_2 = q_2 \bullet t \land \texttt{wfs}_{\texttt{n}}\ s \land \\
\quad (\forall a.\ a \in \texttt{dis_set}\ q_1\ q_2 \Rightarrow \texttt{term_fcs}\ a\ (s \lhd_{\texttt{n}}\ t) \neq \texttt{NONE}) \land \\
\quad \texttt{verify_fcs}\ \mathit{fe}\ s = \texttt{SOME}\ \mathit{ve} \land \texttt{FINITE}\ \mathit{fe} \Rightarrow \\
\qquad \exists \mathit{fex}.\ \texttt{nomunify}\ (s,\mathit{fe})\ t_1\ t_2 = \texttt{SOME}\ (s,\mathit{fex})
\end{array}$$

233. `nvars_measure`

$\vdash v \in \texttt{nvars}\ t \land \neg\texttt{is_Sus}\ t \land \texttt{wfs}_\texttt{n}\ s \Rightarrow$
$\quad \texttt{measure}\ (\texttt{npair_count} \circ \texttt{walk}^*_\texttt{n}\ s)\ (\texttt{Sus}\ []\ v)\ t$

234. `npair_count_apply_pi`

$\vdash \texttt{npair_count}\ (\pi \bullet t) = \texttt{npair_count}\ t$

235. `equiv_depth_eq`

$\vdash fe \vdash t_1 \approx t_2 \Rightarrow \texttt{npair_count}\ t_1 = \texttt{npair_count}\ t_2$

236. `noc_subterm_nequiv`

$\vdash \texttt{oc}_\texttt{n}\ s\ t\ v \land \neg\texttt{is_Sus}\ t \land \texttt{wfs}_\texttt{n}\ s \land \texttt{wfs}_\texttt{n}\ s_2 \Rightarrow$
$\quad \neg(fe \vdash s_2 \lhd_\texttt{n} \texttt{Sus}\ \pi\ v \approx s_2 \lhd_\texttt{n} s \lhd_\texttt{n} t)$

237. `term_fcs_equiv_NONE`

$\vdash \texttt{term_fcs}\ a\ t_1 = \texttt{NONE} \land fe \vdash t_1 \approx t_2 \land \texttt{term_fcs}\ a\ t_2 = \texttt{SOME}\ fcs \Rightarrow$
$\quad \neg(fcs \subseteq fe)$

238. `term_fcs_apply_pi_NONE`

$\vdash \texttt{term_fcs}\ a\ t = \texttt{NONE} \iff \texttt{term_fcs}\ (\pi \bullet a)\ (\pi \bullet t) = \texttt{NONE}$

239. `term_fcs_nwalkstar_NONE`

$\vdash \texttt{wfs}_\texttt{n}\ s \land \texttt{term_fcs}\ a\ t = \texttt{SOME}\ fcs \land \texttt{term_fcs}\ a\ (s \lhd_\texttt{n} t) = \texttt{NONE} \Rightarrow$
$\quad \texttt{verify_fcs}\ fcs\ s = \texttt{NONE}$

240. `verify_fcs_iter_SUBMAP_exists`

$\vdash \texttt{verify_fcs}\ fe\ s = \texttt{SOME}\ ve \land \texttt{wfs}_\texttt{n}\ sx \land s \sqsubseteq sx \land \texttt{FINITE}\ fe \land$
$\quad \texttt{verify_fcs}\ ve\ sx = \texttt{SOME}\ vex \Rightarrow$
$\quad\quad \exists fex.\ \texttt{verify_fcs}\ fe\ sx = \texttt{SOME}\ fex$

241. `equiv_nomunify` (Theorem 8)

$\vdash fe_2 \vdash s_2 \lhd_\texttt{n} s \lhd_\texttt{n} t_1 \approx s_2 \lhd_\texttt{n} s \lhd_\texttt{n} t_2 \land \texttt{wfs}_\texttt{n}\ s_2 \land \texttt{wfs}_\texttt{n}\ s \Rightarrow$
$\quad \exists sx.$
$\quad\quad \forall fe.$
$\quad\quad\quad \texttt{FINITE}\ fe \land \texttt{verify_fcs}\ fe\ sx \neq \texttt{NONE} \Rightarrow$
$\quad\quad\quad\quad \exists fex.\ \texttt{nomunify}\ (s,fe)\ t_1\ t_2 = \texttt{SOME}\ (sx,fex)$

A.16 mKSyntax

(Omitted.)

A.17 sbag

(Omitted)

A.18 mKAbs

242. `sat_rules` (see Definition 41 on 67)

243. `wf_conjunct_def`

```
wf_cjnt (denv,env,gexpr) ⇐⇒
  wf_denv denv ∧ wf_gexpr (ARITIES denv) (FDOM env) gexpr
```

244. `wf_disjunct_def`

```
wf_djnt cf (s,cs) ⇐⇒
  wfs s ∧ sbag_wf cf cs ∧ sbag_every cf wf_cjnt cs
```

245. `wf_state_def`

```
wf_state df cf state ⇐⇒
  sbag_wf df state ∧ sbag_every df (wf_djnt cf) state
```

246. `denv_vars_def`

```
denv_vars denv =
  BIGUNION
    {rangevars ce |
     ∃name cd d f g. DLOOKUP denv name = SOME (cd,ce,d,f,g)}
```

247. `conjunct_vars_def`

```
conjunct_vars (d,e,g) = rangevars e ∪ denv_vars d
```

248. `disjunct_vars_def`

```
disjunct_vars cf (s,cs) =
  substvars s ∪ BIGUNION {conjunct_vars c | sbag_mem cf c cs}
```

249. `denv_vars_DEMPTY`

```
⊢ denv_vars DEMPTY = {}
```

250. `rangevars_FEMPTY`

```
⊢ rangevars FEMPTY = {}
```

251. `substvars_FEMPTY`

```
⊢ substvars FEMPTY = {}
```

252. `rangevars_FUPDATE`

⊢ `rangevars` (s `|+` (x,y)) = `rangevars` (s `\\` x) ∪ `vars` y

253. `substvars_FUPDATE`

⊢ `substvars` (s `|+` (x,y)) = x `INSERT` `substvars` (s `\\` x) ∪ `vars` y

254. `rangevars_DOMSUB_SUBSET`

⊢ `rangevars` (e `\\` x) ⊆ `rangevars` e

255. `substvars_DOMSUB_SUBSET`

⊢ `substvars` (s `\\` x) ⊆ `substvars` s

256. `FINITE_denv_vars`

⊢ `FINITE` (`denv_vars` d)

257. `FINITE_conjunct_vars`

⊢ `FINITE` (`conjunct_vars` c)

258. `FINITE_disjunct_vars`

⊢ `is_sbag` f ∧ `sbag_wf` f cs ⇒ `FINITE` (`disjunct_vars` f (s,cs))

259. `step_via_def`

```
step_via df cf state (s,cs) (SOME s') cont ⟺
  sbag_null cf cs ∧ s' = s ∧ df.replace state [(s,cs)] [] cont
step_via df cf state (s,cs) NONE cont ⟺
  ∃args ccs ccs1 ccs2 cd ce d def defs e e1 e2 f g g1 g2 ls n name
     sx v' x.
    cf.replace cs [(d,e,Conj g1 g2)] [(d,e,g1); (d,e,g2)] ccs ∧
    df.replace state [(s,cs)] [(s,ccs)] cont ∨
    cf.replace cs [(d,e,Disj g1 g2)] [(d,e,g1)] ccs1 ∧
    cf.replace cs [(d,e,Disj g1 g2)] [(d,e,g2)] ccs2 ∧
    df.replace state [(s,cs)] [(s,ccs1); (s,ccs2)] cont ∨
    n ∉ disjunct_vars cf (s,cs) ∧
    cf.replace cs [(d,e,LetVar v' g)] [(d,e |+ (v',Var n),g)]
      ccs ∧ df.replace state [(s,cs)] [(s,ccs)] cont ∨
    cf.replace cs [(d,e,LetVal1 (v',x) g)]
      [(d,e |+ (v',evalex e x),g)] ccs ∧
    df.replace state [(s,cs)] [(s,ccs)] cont ∨
    cf.replace cs [(d,e,Let1 def g)]
      [(DUPDATE d e (NonRec def),e,g)] ccs ∧
    df.replace state [(s,cs)] [(s,ccs)] cont ∨
    cf.replace cs [(d,e,LetRec1 ls g)]
```

```
    [(DUPDATE d e (Rec (FEMPTY |++ ls)),e,g)] ccs ∧
  df.replace state [(s,cs)] [(s,ccs)] cont ∨
  cf.replace cs [(d,e,Call "Eqv" [e1; e2])] [] ccs ∧
  DLOOKUP d "Eqv" = NONE ∧
  unify s (evalex e e1) (evalex e e2) = SOME sx ∧
  df.replace state [(s,cs)] [(sx,ccs)] cont ∨
  cf.replace cs [(d,e,Call "Eqv" [e1; e2])] [] ccs ∧
  DLOOKUP d "Eqv" = NONE ∧
  unify s (evalex e e1) (evalex e e2) = NONE ∧
  df.replace state [(s,cs)] [] cont ∨
  cf.replace cs [(d,e,Call name args)]
    [((case defs of
          Rec v → DUPDATE cd ce defs
        ‖ NonRec v1 → cd),call_env ce f (evalex e) args,g)] ccs ∧
  DLOOKUP d name = SOME (cd,ce,defs,f,g) ∧
  df.replace state [(s,cs)] [(s,ccs)] cont
```

260. step_def

```
step df cf state ans cont ⟺
  ∃ dis. step_via df cf state dis ans cont
```

261. step_rules (see Definition 43 on page 70)

262. DLOOKUP_wf_denv

```
⊢ wf_denv denv ∧
  DLOOKUP denv name = SOME (cdenv,cenv,cdefs,fmls,body) ⇒
    wf_denv (DUPDATE cdenv cenv cdefs) ∧
    wf_gexpr
      (case cdefs of
         Rec fm → LENGTH ∘ FST ∘ fm  ARITIES cdenv
       ‖ NonRec v1 → ARITIES cdenv) (FDOM cenv ∪ set fmls) body
```

263. DLOOKUP_FLOOKUP_ARITIES

```
⊢ wf_denv denv ∧
  DLOOKUP denv name = SOME (cdenv,cenv,cdefs,fmls,body) ⇒
    ARITIES denv ' name = SOME (LENGTH fmls)
```

264. wf_state_push

```
⊢ is_sbag df ∧ wf_state df cf state ∧ EVERY (wf_djnt cf) l2 ∧
  df.replace state l1 l2 cont ⇒
    wf_state df cf cont
```

265. wf_disjunct_push

```
⊢ is_sbag cf ∧ wf_djnt cf (s,cs) ∧ EVERY wf_cjnt l2 ∧
  cf.replace cs l1 l2 ccs ⇒
    wf_djnt cf (s,ccs)
```

266. `wf_state_wf_disjunct`

$\vdash$ `is_sbag` df $\land$ `is_sbag` cf $\land$ `wf_state` df cf $state$ $\land$
df.`replace` $state$ l_1 l_2 $cont$ $\land$ `MEM` d l_1 $\Rightarrow$
`wf_djnt` cf d

267. `FDOM_call_env`

$\vdash$ `LENGTH` $fmls$ $=$ `LENGTH` $args$ $\Rightarrow$
`FDOM` (`call_env` ce $fmls$ $eval$ $args$) $=$ `FDOM` ce $\cup$ `set` $fmls$

268. `wf_state_step`

$\vdash$ `is_sbag` df $\land$ `is_sbag` cf $\land$ `wf_state` df cf $state$ $\land$
`step` df cf $state$ ans $cont$ $\Rightarrow$
`wf_state` df cf $cont$ $\land$ $\forall sx.\ ans$ $=$ `SOME` sx $\Rightarrow$ `wfs` sx

269. `stream_of_def`

`stream_of` R $state$ $llist$ $\iff$
$\exists P.$
($\forall state\ llist.$
P $state$ $llist$ $\Rightarrow$
($\forall ans\ cont.\ \neg R\ state\ ans\ cont$) $\land$ $llist = [||]$ $\lor$
$\exists ans\ cont\ rest.$
$R\ state\ ans\ cont \land P\ cont\ rest \land llist = ans{:::}rest$) $\land$
P $state$ $llist$

270. `stream_of_rules` (see Definition 44 on page 72)

271. `stream_of_coind`

$\vdash$ ($\forall state\ llist.$
$P\ state\ llist \Rightarrow$
($\forall ans\ cont.\ \neg R\ state\ ans\ cont$) $\land$ $llist = [||]$ $\lor$
($\exists ans\ cont\ rest.$
$R\ state\ ans\ cont \land P\ cont\ rest \land llist = ans{:::}rest$) $\lor$
$\exists ans\ cont\ rest.$
$R\ state\ ans\ cont$ $\land$ `stream_of` $R\ cont\ rest$ $\land$
$llist = ans{:::}rest$) $\Rightarrow$
$\forall state\ llist.\ P\ state\ llist \Rightarrow$ `stream_of` $R\ state\ llist$

272. `stream_of_exists` (see Lemma 41 on page 73)

A.19 mKSoundness

(Omitted.)

A.20 mKImp

(Omitted.)

Bibliography

Aït-Kaci, H. 1991. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press. (p. 82)

Andréka, H. and Németi, I. 1980. The generalized completeness of horn predicate-logic as a programming language. *Acta Cybern. 4*, 3–10. (p. 7)

Aydemir, B. E., Bohannon, A., Fairbairn, M., Foster, J. N., Pierce, B. C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., and Zdancewic, S. 2005. Mechanized metatheory for the masses: the POPLmark challenge. In J. Hurd and T. F. Melham Eds., *TPHOLs*, Volume 3603 of *Lecture Notes in Computer Science* (2005), pp. 50–65. Springer. (p. 2)

Baader, F. and Nipkow, T. 1998. *Term Rewriting and All That.* Cambridge University Press. (p. 44)

Baader, F. and Snyder, W. 2001. Unification theory. In J. A. Robinson and A. Voronkov Eds., *Handbook of Automated Reasoning*, pp. 445–532. Elsevier and MIT Press. (pp. 23, 35)

Byrd, W. E. 2009. *Relational Programming in miniKanren: Techniques, Applications, and Implementations.* PhD thesis, Indiana University. (pp. 2, 6, 84)

Byrd, W. E. and Friedman, D. P. 2007. alphaKanren: A fresh name in nominal logic programming languages. *Scheme and Functional Programming*. (pp. 48, 58)

Calvès, C. and Fernández, M. 2008. A polynomial nominal unification algorithm. *Theor. Comput. Sci. 403*, 2-3, 285–306. (p. 82)

Cheney, J. and Urban, C. 2004. alpha-Prolog: A logic programming language with names, binding and α-equivalence. In B. Demoen and V. Lifschitz Eds., *ICLP*, Volume 3132 of *Lecture Notes in Computer Science* (2004), pp. 269–283. Springer. (p. 48)

Cheney, J. and Urban, C. 2008. Nominal logic programming. *ACM Trans. Program. Lang. Syst. 30*, 5. (p. 48)

Cheney, J. R. 2004. *Nominal Logic Programming.* PhD thesis, Cornell University. (p. 48)

de Moura, F., Galdino, A. L., Avelar, A. B., and Ayala-Rincon, M. 2010. Verification of the completeness of unification algorithms à la robinson. In A. Dawar Ed., *Proceedings of the 17th Workshop on Logic, Language, Information and Computation* (2010). To appear. (p. 82)

Friedman, D. P., Byrd, W. E., and Kiselyov, O. 2005. *The Reasoned Schemer.* The MIT Press. (pp. 2, 6)

FRIEDMAN, D. P. AND WISE, D. S. 1976. CONS should not evaluate its arguments. In *ICALP* (1976), pp. 257–284. (p. 83)

GORDON, M. 2000. From LCF to HOL: a short history. In G. D. PLOTKIN, C. STIRLING, AND M. TOFTE Eds., *Proof, Language, and Interaction: Essays in Honour of Robin Milner* (2000), pp. 169–186. The MIT Press. (p. 8)

HARRISON, J. 1996. Proof style. In E. GIMÉNEZ AND C. PAULIN-MOHRING Eds., *TYPES*, Volume 1512 of *Lecture Notes in Computer Science* (1996), pp. 154–172. Springer. (p. 17)

HODER, K. AND VORONKOV, A. 2009. Comparing unification algorithms in first-order theorem proving. In B. MERTSCHING, M. HUND, AND M. Z. AZIZ Eds., *KI*, Volume 5803 of *Lecture Notes in Computer Science* (2009), pp. 435–443. Springer. (p. 26)

HORN, A. 1951. On sentences which are true of direct unions of algebras. *J. Symb. Log. 16*, 1, 14–21. (p. 7)

KAPLAN, H. 2004. Persistent data structures. In D. P. MEHTA AND S. SAHNI Eds., *Handbook on Data Structures and Applications*, pp. 31–1–31–27. Chapman & Hall/CRC. (p. 83)

KISELYOV, O., BYRD, W. E., FRIEDMAN, D. P., AND CHIEH SHAN, C. 2008. Pure, declarative, and constructive arithmetic relations (declarative pearl). In J. GARRIGUE AND M. V. HERMENEGILDO Eds., *FLOPS*, Volume 4989 of *Lecture Notes in Computer Science* (2008), pp. 64–80. Springer. (p. 6)

KISELYOV, O., CHIEH SHAN, C., FRIEDMAN, D. P., AND SABRY, A. 2005. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In O. DANVY AND B. C. PIERCE Eds., *ICFP* (2005), pp. 192–203. ACM. (pp. 76, 83)

KNIGHT, K. 1989. Unification: A multidisciplinary survey. *ACM Comput. Surv. 21*, 1, 93–124. (p. 35)

KOWALSKI, R. A. 1974. Predicate logic as programming language. In *IFIP Congress* (1974), pp. 569–574. (p. 2)

LEVY, J. AND VILLARET, M. 2008. Nominal unification from a higher-order perspective. In A. VORONKOV Ed., *RTA*, Volume 5117 of *Lecture Notes in Computer Science* (2008), pp. 246–260. Springer. (p. 82)

LLOYD, J. W. 1987. *Foundations of Logic Programming, 2nd Edition*. Springer. (p. 82)

MANNA, Z. AND WALDINGER, R. J. 1981. Deductive synthesis of the unification algorithm. *Sci. Comput. Program. 1*, 1-2, 5–48. (p. 43)

MARTELLI, A. AND MONTANARI, U. 1982. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst. 4*, 2, 258–282. (pp. 42, 43, 45, 83)

MCBRIDE, C. 2003. First-order unification by structural recursion. *J. Funct. Program. 13*, 6, 1061–1075. (p. 82)

Milner, R., Tofte, M., Harper, R., and MacQueen, D. 1997. *The Definition of Standard ML, Revised Edition.* The MIT Press. (p. 8)

Near, J. P., Byrd, W. E., and Friedman, D. P. 2008. αleanTAP: A declarative theorem prover for first-order classical logic. In M. G. de la Banda and E. Pontelli Eds., *ICLP*, Volume 5366 of *Lecture Notes in Computer Science* (2008), pp. 238–252. Springer. (p. 58)

Norrish, M. and Slind, K. 1998. HOL4 manuals. `http://hol.sourceforge.net`. (p. 7)

Norrish, M. and Slind, K. 2002. A thread of HOL development. *Comput. J. 45*, 1, 37–45. (p. 8)

Okasaki, C. 1998. *Purely Functional Data Structures.* Cambridge University Press. (p. 83)

Paterson, M. and Wegman, M. N. 1978. Linear unification. *J. Comput. Syst. Sci. 16*, 2, 158–167. (p. 45)

Paulson, L. C. 1985. Verifying the unification algorithm in LCF. *Sci. Comput. Program. 5*, 2, 143–169. (pp. 20, 43, 81)

Paulson, L. C. 1997. Mechanizing coinduction and corecursion in higher-order logic. *J. Log. Comput. 7*, 2, 175–204. (p. 72)

Pelletier, F. J. 1986. Seventy-five problems for testing automatic theorem provers. *J. Autom. Reasoning 2*, 2, 191–216. (p. 58)

Pitts, A. M. 2003. Nominal logic, a first order theory of names and binding. *Inf. Comput. 186*, 2, 165–193. (p. 47)

Reynolds, J. C. 1998. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation 11*, 4, 363–397. (p. 78)

Robinson, J. A. 1965. A machine-oriented logic based on the resolution principle. *J. ACM 12*, 1, 23–41. (p. 82)

Ruiz-Reina, J.-L., Martín-Mateos, F.-J., Alonso, J.-A., and Hidalgo, M.-J. 2006. Formal correctness of a quadratic unification algorithm. *J. Autom. Reasoning 37*, 1-2, 67–92. (pp. 82, 83)

Slind, K. 1999. *Reasoning about Terminating Functional Programs.* PhD thesis, Technischen Universität München. (pp. 9, 20, 43, 82)

Slind, K. and Norrish, M. 2008. A brief overview of HOL4. In O. A. Mohamed, C. Muñoz, and S. Tahar Eds., *TPHOLs*, Volume 5170 of *Lecture Notes in Computer Science* (2008), pp. 28–32. Springer. (p. 8)

Sperber, M., Dybvig, R. K., Flatt, M., van Straaten, A., and Matthews, J. 2009. Revised[6] report on the algorithmic language scheme. *Journal of Functional Programming 19*, Supplement S1, 1–301. (p. 3)

Spivey, J. M. 2000. Combinators for breadth-first search. *J. Funct. Program. 10*, 4, 397–408. (p. 83)

Urban, C. 2004. Nominal unification. `http://www4.in.tum.de/~urbanc/Unification/`. (p. 82)

Urban, C., Pitts, A. M., and Gabbay, M. 2004. Nominal unification. *Theor. Comput. Sci. 323*, 1-3, 473–497. (pp. 48, 50, 51, 52, 53, 82)

Warren, D. H. D. 1983. An abstract prolog instruction set. Technical Note 309 (Oct.), SRI International, Menlo Park, CA. (p. 82)

Wenzel, M. 1999. Isar - a generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry Eds., *TPHOLs*, Volume 1690 of *Lecture Notes in Computer Science* (1999), pp. 167–184. Springer. (p. 17)