Technical Report

UCAM-CL-TR-772
ISSN 1476-2986

Number 772

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

An executable meta-language for inductive definitions with binders

Matthew R. Lakin

March 2010

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

© 2010 Matthew R. Lakin

This technical report is based on a dissertation submitted March 2010 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Queen's College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

An executable meta-language for inductive definitions with binders

Matthew R. Lakin

A testable prototype can be invaluable for identifying bugs during the early stages of language development. For such a system to be useful in practice it should be quick and simple to generate prototypes from the language specification.

This dissertation describes the design and development of a new programming language called αML, which extends traditional functional programming languages with specific features for producing correct, executable prototypes. The most important new features of αML are for the handling of names and binding structures in user-defined languages. To this end, αML uses the techniques of nominal sets (due to Pitts and Gabbay) to represent names explicitly and handle binding correctly up to α-renaming. The language also provides built-in support for constraint solving and non-deterministic search.

We begin by presenting a generalised notion of systems defined by a set of schematic inference rules. This is our model for the kind of languages that might be implemented using αML. We then present the syntax, type system and operational semantics of the αML language and proceed to define a sound and complete embedding of schematic inference rules. We turn to program equivalence and define a standard notion of operational equivalence between αML expressions and use this to prove correctness results about the representation of data terms involving binding and about schematic formulae and inductive definitions.

The fact that binding can be represented correctly in αML is interesting for technical reasons, because the language dispenses with the notion of globally distinct names present in most systems based on nominal methods. These results, along with the encoding of inference rules, constitute the main technical payload of the dissertation. However, our approach complicates the solving of constraints between terms. Therefore, we develop a novel algorithm for solving equality and freshness constraints between nominal terms which does not rely on standard devices such as swappings and suspended permutations. Finally, we discuss an implementation of αML, and conclude with a summary of the work and a discussion of possible future extensions.

Contents

1 Introduction **13**

1.1 The language design process . . . 13
1.2 Names and binding . . . 17
1.3 Existing approaches to binding . . . 20
1.4 The novelty of αML . . . 28
1.5 Dissertation overview . . . 30
1.6 On collaboration . . . 31

2 α-inductive definitions **33**

2.1 Signatures and equality types . . . 33
2.2 Ground trees and α-equivalence classes . . . 35
2.3 Syntax of α-inductive definitions . . . 36
2.4 Semantics of α-inductive definitions . . . 42
2.5 α-inductive definitions and equivariance . . . 47

3 αML **51**

3.1 Language overview . . . 51
3.2 αML syntax . . . 52
3.3 Static semantics . . . 54
3.4 α-tree constraint problems . . . 59
3.5 Operational semantics . . . 63
3.6 Embedded functional programming language . . . 71
3.7 Type safety . . . 71

4 α-inductive definitions in αML **75**

4.1 α-inductive definitions as αML recursive functions . . . 75
4.2 Embedded constraint logic programming language . . . 76
4.3 Soundness and completeness . . . 78
4.4 Example definition . . . 85

5 Contextual equivalence **87**

5.1 Definition of operational equivalence . . . 87
5.2 Expression relations . . . 88
5.3 CIU theorem . . . 90
5.4 Correctness of data representation . . . 91
5.5 Contextual equivalence of formulae . . . 98
5.6 Operational equivalence with finite failure . . . 98
5.7 Fresh name generation . . . 100

6 Constraint solving **103**
6.1 Constraint transformation . . . 103
6.2 Soundness and completeness of transformations . . . 106
6.3 Towards a decision procedure . . . 108
6.4 A tractable subproblem . . . 112

7 Implementation **117**
7.1 Interpreter overview . . . 117
7.2 Extended αML . . . 119
7.3 Implementing the constraint solver . . . 122

8 Conclusions and future work **125**
8.1 Future work . . . 126
8.2 Final remarks . . . 132

Appendices **141**

A Proof of compatibility **143**
A.1 Proof outline . . . 143
A.2 Proof of compatibility: `let` bindings . . . 145
A.3 Proof of compatibility: recursive functions . . . 147
A.4 Summary of compatibility . . . 151

B Contextual equivalence of formulae **153**
B.1 Contextually equivalent formulae have the same semantics . . . 153
B.2 Formulae with the same semantics are contextually equivalent . . . 156

C Implementation details **159**
C.1 The bytecode machine . . . 159
C.2 Compiling αML expressions . . . 161

D Using the interpreter **165**
D.1 An example program: System F . . . 165
D.2 Interacting with the toplevel . . . 169

List of Figures

1.1 Example inference rules for call-by-name λ-calculus . . . 14
1.2 Language design cycles, after (Boehm, 1986) . . . 15

2.1 Example nominal signature $\mathcal{F}$ for System F . . . 35
2.2 Typing rules for schematic patterns . . . 37
2.3 Typing rules for schematic constraints and formulae . . . 40
2.4 Example α-inductive definitions for the System F type system . . . 42
2.5 Formula satisfaction rules . . . 46

3.1 αML values, expressions and constraints . . . 53
3.2 Typing relation for αML values, constraints and expressions . . . 57
3.3 Pathological example of ML type inference . . . 63
3.4 Small-step operational semantics for αML . . . 67

4.1 Formula reduction . . . 77

5.1 Compatible refinement $\widehat{\mathcal{E}}$ of an expression relation $\mathcal{E}$ . . . 89
5.2 Extension of compatible refinement to frame stacks and constraint problems . . . 89
5.3 Tree translation rules . . . 93

6.1 Constraint transformation rules . . . 104
6.2 Narrowing rules . . . 106

7.1 Interpreter structure . . . 117
7.2 Examples of constraint representation . . . 123

8.1 Encoding of nominal unification terms . . . 130

C.1 Informal semantics of bytecode instructions . . . 160
C.2 Compilation function . . . 162

For Aidan, and for my parents

Acknowledgements

I must begin by thanking my supervisor, Andrew Pitts. I have worked with Andy for almost five years and he has been a constant source of encouragement and excellent advice throughout that time. His assistance with proof-reading has been invaluable—this dissertation exists primarily because of him. I would also like to thank Robin Walker, my Director of Studies at Queens', for permitting me to switch from Physics to Computer Science and for providing the wine at Wednesday formals.

My research studentship was funded by the UK Engineering and Physical Sciences Research Council (EPSRC grant EP/D000459/1) through the Computational Applications of Nominal Sets (CANS) project. I am grateful to EPSRC and to the University of Cambridge Computer Laboratory for financial support throughout my work, and to the organisers and participants of the CANS project for the vibrant exchange of ideas at the project meetings. My office mates, David Turner and in particular Ranald Clouston, deserve a special mention for putting up with my various annoying habits. I am also indebted to James Cheney and François Pottier for helpful comments and discussions as well as to an anonymous reviewer from PPDP 2008, whose insightful comments helped to shape the final direction of my research. I would like to thank my examiners, Alan Mycroft and Paul Levy, for their constructive advice on the final version of this dissertation.

During the course of my work I was fortunate enough to spend time as a visiting researcher in Germany and the United States. I am grateful to Christian Urban and the rest of the Nominal Methods group at the Technische Universität München for teaching the basics of Nominal Isabelle, and espcially to Daniel and Mary Friedman for welcoming me into their home during my visit to Indiana University. I would also like to thank William Byrd for looking after me during my stay in Bloomington.

I am forever indebted to my parents and family for their love and support, for giving me the opportunities to continue in education and for buying me my first computer for Christmas. Thank you for everything.

Last but not least I owe so much to my wife Aidan, for always being there and for always believing in me. I love you to the Delta Quadrant and back!

Chapter 1

Introduction

"The value of a prototype is in the education it gives you, not in the code itself."

—A. Cooper

My thesis is as follows:

> *Rapid and correct prototyping of programming languages is possible using the executable meta-language αML.*

This dissertation will provide evidence to support my thesis, gained from the design and theoretical development of the αML language. αML is intended as a tool for programming language designers, and allows easy generation of executable code which models the behaviour of the language in question.

Reliable software requires reliable programming languages. The need for reliable software in everyday life is self-evident, from the autopilot program on a commercial jet aircraft to the control program for a nuclear reactor or missile defence system. However, the need for reliable *programming languages* is arguably more urgent—bugs in a language design or compiler infect all programs written in that language, which might include yet more compilers. To quote Tony Hoare's Turing Award lecture (Hoare, 1981):

> *An unreliable programming language generating unreliable programs constitutes a far greater risk to our environment and to our society than unsafe cars, toxic pesticides or accidents at nuclear power stations.*

Therefore, it is vital that the languages produced by language designers, and the tools that they use, are robust and correct.

1.1 The language design process

Some sixty years after the construction of the first programmable computers the process of programming language design is still something of a black art. Papers exist from the early days of the computer industry (Hoare, 1973; Wirth, 1974) which describe broad criteria such as simplicity, efficiency and readability, but little (if any) formal research into effective language design procedures has been carried out. The same is not true of software engineering and the design of software systems: see (Glass et al., 2002) for an empirical analysis of the scale and diversity

Language syntax: $t ::= x \mid t\,t \mid \lambda x.\, t.$

Typing rules:

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \frac{\Gamma \vdash t : T' \to T \qquad \Gamma \vdash t' : T'}{\Gamma \vdash t\,t' : T} \qquad \frac{\Gamma, x : T \vdash t : T' \qquad x \notin dom(\Gamma)}{\Gamma \vdash \lambda x.\, t : T \to T'}$$

Operational semantics:

$$\frac{t_1 \longrightarrow t_1'}{t_1\, t_2 \longrightarrow t_1'\, t_2} \qquad \frac{}{(\lambda x.\, t)\, t' \longrightarrow t[t'/x]}$$

Figure 1.1: Example inference rules for call-by-name λ-calculus

of software engineering literature. [1] One promising approach to managing the complexity inherent in language design is the adoption of formal methods. These involve modelling the abstract syntax of the language of interest, the object-language, in some host language, the meta-language. Meta-languages are specifically designed for manipulating the syntactic structure of other programs. Examples of meta-programming include theorem provers (which manipulate the syntax of logical sentences) and compilers (which translate source to target languages).

Properties of programming languages are often expressed as judgements $\mathtt{J}$ about object-language terms. Examples include typing judgements ($\Gamma \vdash t : T$) and operational semantics ($t \longrightarrow t'$). These are typically defined using inference rules, which take the form

$$\frac{\mathtt{J}_1 \ \cdots \ \mathtt{J}_n \quad \psi_1 \ \cdots \ \psi_m}{\mathtt{J}}$$

where the ψ_i are logical statements known as side-conditions, whose function is to prevent certain applications of the rule. $\mathtt{J}$ is the conclusion of this rule, and the judgements and side-conditions above the line are the premises. Figure 1.1 presents an example of the type system and operational semantics of a small language (the call-by-name λ-calculus) defined using inference rules. It is worth noting that the specification in Figure 1.1 is incomplete, because $t[t'/x]$ is not defined.

Inference rules such as those from Figure 1.1 are *schematic* in the sense that they provide a template for a collection of specific inferences, derived by consistently instantiating the variables. Inference rules can be read in two ways:

- **top-down:** if the premises all hold under some instantiation of the variables then the conclusion must also hold under that instantiation.
- **bottom-up:** to prove that the conclusion holds under some instantiation, we must show that the premises all hold under that instantiation.

A derivation of a judgement is a tree of rule applications which results in that judgement appearing in the conclusion at the very bottom, such as the following derivation of the typing judgement $\{z : \mathsf{int}\} \vdash \lambda x.\, x : \mathsf{int} \to \mathsf{int}$.

$$\frac{\dfrac{(x : \mathsf{int}) \in \{z : \mathsf{int}, x : \mathsf{int}\}}{\{z : \mathsf{int}, x : \mathsf{int}\} \vdash x : \mathsf{int}} \qquad x \notin dom(\{z : \mathsf{int}\})}{\{z : \mathsf{int}\} \vdash \lambda x.\, x : \mathsf{int} \to \mathsf{int}} \tag{1.1}$$

[1] It is worth noting, however, that the problem of delivering large software projects is by no means solved!

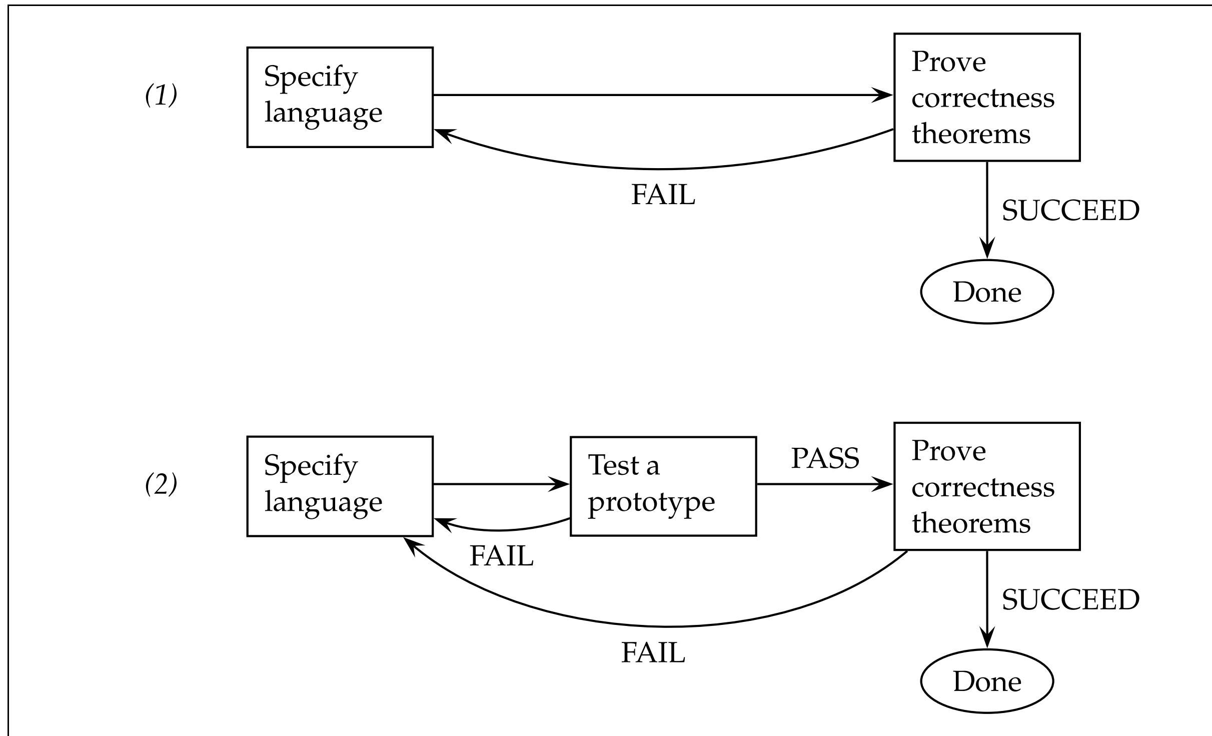


Figure 1.2: Language design cycles, after (Boehm, 1986)

We will concern ourselves largely with the bottom-up reading of inference rules. In order to determine whether $\{z:\mathsf{int}\} \vdash \lambda x.\, x:\mathsf{int} \rightarrow \mathsf{int}$ holds, we must check whether there exists a derivation of the judgement using the rules from Figure 1.1, i.e. the derivation (1.1). We will refer to this kind of computation as proof-search.

The specification of a language's type system or operational semantics as a set of inference rules is just the first step in the language design process. The designer is then faced with a choice: they can jump in and write an implementation of the language by hand, or wait and do more work on the language specification. It may be beneficial to encode the inference rules in some meta-language at this stage, so that they can be manipulated with the help of the computer. This can protect against human errors and bugs in the specification.

The gold standard for language design is a formal, mechanised proof of a correctness theorem. This provides a fairly high degree of assurance that the language design is correct (in some sense). However, the barrier to entry for most automated theorem provers is quite high, and the amount of effort needed to prove non-trivial properties of non-trivial languages can be considerable: a user's initial forays into mechanised theorem proving are described with dry wit in (Benton, 2006). Nonetheless, larger formalisations are possible with a larger investment of time and significant experience: for example, Leroy has developed a certified compiler from a substantial subset of C into PowerPC assembler code using the Coq theorem prover (Leroy, 2009) and a formalisation of x86 multiprocessor machine code in HOL is described in (Sarkar et al., 2009).

Although it is an excellent means of detecting bugs and ambiguities in specifications, the time investment required to fully mechanise proofs is significant. This means that a language design cycle such as *(1)* from Figure 1.2 would take a long time to complete. A possible compromise is offered by *testing*—for most languages, it is possible to isolate a set of particular terms as test cases, which can verify certain behaviours of the system. This gives a reduced level of confidence in the language design, but at a much lower cost. This is illustrated as *(2)*

in Figure 1.2. The lower cost of testing means that it can be done earlier in the design process and guide the refinement of the language design. Hopefully the testing process will mean that there are fewer unsuccessful attempts to prove the correctness theorems.

In order to test, however, we need an executable prototype of the language design. Ideally, it should be very easy to produce this prototype from the language specification, to minimise the barrier to entry. We therefore motivate the development of the αML meta-language as part of a tool-chain to help language designers produce more reliable programming languages. αML code can be automatically generated from a high-level specification provided as a collection of inference rules, and used to test the behaviour of the system in various situations. The language works by running a proof-search procedure over the inference rules of the inductive definition. This activity has been variously referred to in the literature as *prototyping* and *animating*. This terminology suggests it is only useful for modelling operational semantics, but in fact any inductively-defined relation can be "animated", for example type systems.

There is already considerable movement towards the adoption of formal methods, in particular in the academic community. The POPLMARK challenge (Aydemir et al., 2005) has provided a focus for efforts to encode, prove theorems about and produce a prototype implementation of a small calculus (a polymorphic λ-calculus with records and subtyping). At this point it is useful to informally distinguish between two flavours of language design effort:

- **large-scale languages** which are intended for widespread industrial use for general programming tasks, such as C and Java. Large-scale industrial programming languages are often designed by international standards committees, for example ANSI C. However, this approach can lead to delays, bugs and feature-creep—see (Hoare, 1981) for a fascinating insight.

- **smaller languages, domain-specific languages and calculi** which tend to be designed for specific purposes by individuals or small teams. These are often used as research vehicles by the academic community but domain-specific languages are increasingly finding commercial use in various application domains, for example the financial sector (Peyton-Jones et al., 2000; Frankau et al., 2009). The application of formal methods is more feasible for these smaller languages.

It is worth pointing out that there *are* examples of large-scale language design efforts which have employed pre-hoc specification. Standard ML, for example, has a formal mathematical specification, which was initially published as (Milner et al., 1990). However, in the absence of machine-verified correctness proofs, various errors and ambiguities in the specification emerged over time (Kahrs, 1993), which were later corrected in the revised definition (Milner et al., 1997). Another effort worthy of note is by a standards committee: the ongoing standardisation of Javascript uses ML as its specification language (Herman and Flanagan, 2007).

The development of αML is a prime example of the informal language design process described here. Over a period of several years, the syntax and semantics of the language changed quite dramatically before finally stabilising in the form described in this dissertation. The lack of a readily-available prototyping mechanism meant that several toy implementations had to be hand-written, which slowed the development of the language considerably. This highlights the difficulty inherent in designing a meta-language for encoding object-languages and their semantics: it must be general enough to encode a wide range of interesting systems but provide specific features to make the encoding simple and efficient.

In order to provide a flavour of the evolution of αML, this dissertation contains some remarks which are earmarked as **historical notes**. These chart the evolution of various features of the language, from its conception almost five years ago through numerous revisions to the

language described here. To avoid confusion, the reader should assume that the features discussed in these notes are deprecated.

1.2 Names and binding

The issues of names and name-binding are a major difficulty when formalising programming language meta-theory.

Names are ubiquitous throughout logic, mathematics and computer science. We concern ourselves here with pure names, or names with no internal structure. The concept of a pure name is introduced in (Needham, 1993). That paper considered a name as a "bit-pattern that is an identifier", but we take a more abstract view of names as mathematical entities. We assume only that two names can be compared for equality. This allows us to use them as pointers or to represent unknown quantities. For example, in the mathematical expression "$y = x + 1$", the names x and y refer to unknown numbers.

A consequence of our abstract view of names is that they do not appear in the syntax of programs. This approach is adopted in Barendregt's seminal text on the λ-calculus (Barendregt, 1984). There, λ-terms are defined as words over an alphabet containing variables v_0, v_1, v_2 etc. (Definition 2.1.1). However, the set Λ of λ-terms is then defined as the least set satisfying the axioms

$$(1)\ \frac{}{x \in \Lambda} \qquad (2)\ \frac{t \in \Lambda}{(\lambda x.\, t) \in \Lambda} \qquad (3)\ \frac{t, t' \in \Lambda}{(t\, t') \in \Lambda}$$

where x in (1) or (2) is an *arbitary variable*. The remainder of Barendregt's book uses only the meta-variables x, y, z, etc., which were introduced in Barendregt's Notation 2.1.2 and range over arbitrary variables. There is no other mention of the real variables v_0, v_1, v_2 etc. This notational convention means that we are not forced to choose a particular variable (v_{42} say) for use in a certain situation.

In fact, most work on programming language semantics goes a step further and insists that the meta-variables x, y, z, $\ldots$ range *permutatively* over the set of names, i.e. if x and y are syntactically different meta-variables then they refer to distinct names. Gabbay refers to this as the **permutative convention** (Gabbay and Mathijssen, 2008). Under the permutative convention, the symbols like x and y that we write down really do correspond to arbitrarily-chosen names. The distinction may seem pedantic, but we beg the reader's indulgence: *one of the main technical contributions of this dissertation is removing the permutative convention.*

The real power of names comes when they may be bound. Name-binding pervades many mathematical systems of interest, and we illustrate a few of these below. We write $\varphi(x)$ to stand for logical sentences that may involve the name x.

$\int_0^1 x + y\, dx$	integration variables,
$\forall x.\ \varphi(x)$	quantifiers in logic,
$\lambda x.\, \lambda y.\, x\, y\, x$	λ-abstraction.

In all these examples there is a binding occurrence of the variable to which the bound occurrences refer. However, only occurrences of the bound name in the lexical scope of the binding refer back to the bound occurrence. For example, in the logical formula

$$(\exists x.\ \varphi(x)) \wedge (\forall x.\ \psi(x))$$

the scope of the first bound x is φ and the scope of the second bound x is ψ.

The main problem for encoding abstract syntax with binders is that the standard notion of equality is not syntactic equality but the larger relation of α-equivalence. Two terms are α-equivalent if they can be made syntactically equal by a *capture-avoiding* renaming of bound names, i.e. one which does not alter the binding structure of the term by bringing more names into the scope of a binder. For example, the following λ-terms are α-equivalent

$$\lambda x.\, x\, y =_\alpha \lambda z.\, z\, y \tag{1.2}$$

because replacing every occurrence of x with z and every occurrence of y with x in $\lambda x.\, x\, y$ produces $\lambda z.\, z\, y$ without changing the binding structure. However, the following terms are *not* α-equivalent

$$\lambda x.\, x\, y \neq_\alpha \lambda y.\, y\, y \tag{1.3}$$

because the binding structure of the two terms is clearly not the same—the y which is free on the left-hand side has been captured on the right-hand side.

Remark 1.2.1 (Using the permutative convention). It is worth pointing out that the truth of (1.2) and (1.3) hinge on our adoption of the *permutative convention* that x, y and z stand for distinct names. $\diamondsuit$

The treatment of binders in informal mathematics hinges on the *Barendregt variable convention* (Barendregt, 1984). In that text we have

> *2.1.12.* CONVENTION. *Terms that are [α-equivalent] are identified. So now we write $\lambda x.\, x = \lambda y.\, y$, etcetera.*
>
> *2.1.13.* VARIABLE CONVENTION. *If $t_1, \ldots, t_n$ occur in a certain mathematical context (e.g. definition, proof) then these in these terms all bound variables are chosen to be different from the free variables.*

which lead to the following moral.

> *2.1.14.* MORAL. *Using conventions 2.1.12 and 2.1.13 one can work with λ-terms in the naïve way.*

This means that when we write a term such as $\lambda x.\, \lambda y.\, x$ we are really referring to an entire α-equivalence class $[\lambda x.\, \lambda y.\, x]_\alpha$. Furthermore, the schematic variables in inference rules such as those presented for the λ-calculus in Figure 1.1 really stand not for λ-terms but for *α-equivalence classes of λ-terms*. A special case is when x is intended to stand for a name: the α-equivalence class of a name n is just the singleton set $\{n\}$. Therefore, variables of name sorts really do refer to arbitrary single names, even under the variable convention.

The derivation (1.1) represents not just a single typing judgement for $\lambda x.\, x$ but a whole family of judgements related by α-conversion:

$$\{z : \mathsf{int}\} \vdash \lambda x.\, x : \mathsf{int} \to \mathsf{int} \qquad \{z : \mathsf{int}\} \vdash \lambda w.\, w : \mathsf{int} \to \mathsf{int} \qquad \{z : \mathsf{int}\} \vdash \lambda q.\, q : \mathsf{int} \to \mathsf{int} \qquad \cdots$$

It is instructive to consider the following judgement, which is a special case of the α-equivalent judgements listed above. Here, the bound variable z matches the variable in the environment.

$$\{z : \mathsf{int}\} \vdash \lambda z.\, z : \mathsf{int} \to \mathsf{int} \tag{1.4}$$

At first glance, it seems that this judgement cannot be derived using the rules from Figure 1.1: a partial derivation would look as follows.

$$\dfrac{\dfrac{\vdots}{\{z : \mathsf{int}, z : \mathsf{int}\} \vdash z : \mathsf{int}} \qquad z \notin \mathit{dom}(\{z : \mathsf{int}\})}{\{z : \mathsf{int}\} \vdash \lambda z.\, z : \mathsf{int} \to \mathsf{int}}$$

At first glance it seems that the "$z \notin dom(\{z : \mathsf{int}\})$" side-condition should fail, because the variable z is clearly in the domain of the typing environment $\{z : \mathsf{int}\}$. However, we are forgetting that λ-terms are identified up to α-equivalence and so this judgement is equivalent to those already established above, such as (1.1). This is an example of the variable convention in practice: we can always rename the bound name to be distinct from any finite set of names in the surrounding context—here, the typing environment. We aim to build this informal reasoning about names into αML.

Use of the variable convention certainly leads to slick informal proofs in the cases involving binders, but things get more difficult if we want a formal, machine-checked proof. A simple example is the weakening lemma in type theory. For the λ-calculus type system defined in Figure 1.1:

> LEMMA (WEAKENING). *If $\Gamma \vdash t : T$ is derivable and $\Gamma' \supseteq \Gamma$ then $\Gamma' \vdash t : T$ is also derivable.*

The proof goes by rule induction over the $\Gamma \vdash t : T$ judgement. The case for the λ rule requires the variable convention: the informal proof goes as follows (paraphrased from (Urban et al., 2007)).

> ***Assume that $\Gamma \vdash t : T$, where $t = \lambda x.\, t_1$ and $T = T_1 \to T_2$.*** *From the typing rule for λ-binders we get that $\Gamma, x : T_1 \vdash t_1 : T_2$ and $x \notin dom(\Gamma)$. Using the variable convention we may assume that $x \notin dom(\Gamma')$, and furthermore $\Gamma', x : T_1 \supseteq \Gamma, x : T_1$. By induction we get $\Gamma', x : T_1 \vdash t_1 : T_2$, and by the typing rule we have that $\Gamma' \vdash \lambda x.\, t_1 : T_1 \to T_2$.*

Formalising this proof is difficult because of the informal use of the variable convention. Implicit assumptions must be formalised: for example we assume that there are always fresh names to choose which do not appear in the surrounding context. Naïve reasoning with the variable convention has been shown to lead to inconsistencies in the context of the Nominal Isabelle theorem prover, for example the "faulty lemma" from (Urban et al., 2007). A mechanisation of this weakening result in Nominal Isabelle is described in Section 5.1 of that paper.

A key operation on bound names is substituting them away, and similar scoping rules apply. If the binder refers to some object, then all (and only) bound names in scope should be substituted with the same object. We write $t'[t/x]$ for the result of substituting the term t for all *free* (i.e. not bound) occurrences of t in t'. For λ-terms, substitution can be defined inductively as follows.

$$x[t/x] = t \qquad y[t/x] = y \qquad (t_1\, t_2)[t/x] = (t_1[t/x])\,(t_2[t/x])$$

$$(\lambda y.\, t')[t/x] = \lambda y.\, (t'[t/x]) \quad \text{where } x \neq y \text{ and } x \notin FV(t).$$

The side-conditions in the binding case can always be satisfied by a suitable capture-avoiding renaming of the bound variable. The first ensures that only free occurrences of x are replaced, and the second preserves the binding structure of the terms. For example, $(\lambda y.\, (y\, x))[y/x]$ results in $\lambda z.\, (z\, y)$, not $\lambda y.\, (y\, y)$.

An overarching theme in the development of programming languages and proof assistants for abstract syntax involving binders is incorporating the variable convention in some sense, thereby modelling "informal practice". These all aim to remove some of the burden of thinking about name-binding issues from the user and transfer it to the machine. As we shall see, αML is no different in this respect.

1.3 Existing approaches to binding

We now present a survey of different approaches to modelling the abstract syntax of object-languages which involve binders.

- **de Bruijn indices (de Bruijn, 1972)** are a well-established technique for representing terms with binding. They are a nameless representation: bound variables are replaced by a natural number index which records the number of λ symbols which are in scope between the bound variable and its binding occurrence. For example, the K combinator $(\lambda x.\, \lambda y.\, x)$ is written as $\lambda\, \lambda\, 2$, and the S combinator $(\lambda x.\, \lambda y.\, \lambda z.\, x\, z\, (y\, z))$ becomes $\lambda\, \lambda\, \lambda\, 3\, 1\, (2\, 1)$.

  This approach is attractive because two α-equivalent terms have the same de Bruijn representation. Therefore they are particularly amenable to manipulation by a computer and are frequently used to represent binding in compiler intermediate languages (indeed, the implementation of αML described in Chapter 7 uses a de Bruijn representation internally). However, de Bruijn indices are less convenient for reasoning by humans, due to delicate issues surrounding the "shifting" of indices when a term is placed into a new context.

- **Higher-order abstract syntax (Pfenning and Elliott, 1988)** (HOAS) is another popular nameless representation. This approach uses a typed λ-calculus as its meta-language. Object-language names are interpreted as λ-calculus variables and the built-in λ-binder is used to encode object-language binders. This is an elegant solution because it sweeps the problem of object-language binding under the carpet.

  HOAS techniques have been used extensively in logical frameworks such as λProlog (Nadathur and Miller, 1988) and Twelf (Pfenning and Schürmann, 1999). As a simple example, the λ-calculus itself can be encoded in a HOAS system as the following type.

  ```
  datatype lam = Lam of lam → lam | App of lam * lam
  ```

  The key point is that the `Lam` constructor takes a function from `lam` to `lam`, which can be thought of as computing the result of applying the λ-abstraction to another term. This means that we get a notion of simple object-language substitution "for free" from meta-level application, so the β-rule for λ-calculus could be implemented as the rule

  $$\texttt{App ((Lam}\, f\texttt{)}\, ,\, t\texttt{)} \longrightarrow f(t).$$

  This is often cited as an advantage of higher-order syntax representations. However, substitution is usually not difficult to define and if one wants an alternative notion of substitution (such as parallel substitution) it must be defined by hand as usual.

  A serious problem for HOAS is that it is difficult to get reasonable-looking induction principles over languages defined as a HOAS type. The problem is caused by the negative occurence of `lam` on the left-hand side of the function type in the `Lam` case. A possible solution is to move to weak HOAS (Despeyroux et al., 1995), where there is a name sort (`var`) and the type declaration above would become

  ```
  datatype lam = Var of var | Lam of var → lam | App of lam * lam
  ```

  where the higher-order function maps variables to λ-terms instead of λ-terms to λ-terms. The negative occurrence of `lam` has disappeared, but capture-avoiding substitution no longer comes for free but must be defined inductively.

  If we implement the λ-calculus using the first type declaration above in a functional programming language then there are many typeable expressions with type `lam` which do not

correspond to a HOAS-encoding of any λ-term. This is because expressions of type `lam` $\rightarrow$ `lam` may perform arbitrary computation in a programming language—these are known as exotic terms. In (Despeyroux et al., 1995) the problem of exotic terms in Coq was overcome by defining an additional validity predicate. In the context of weak HOAS, the validity predicate must ensure that exotic terms which use the binder to perform arbitrary computation must be ruled out.

In order to found a logic programming system on HOAS, we need a unification algorithm to perform resolution. Higher-order unification (Huet, 1975) unifies typed λ-terms up to $\alpha\beta\eta$-conversion, but this is undecidable (Goldfarb, 1981). However, a decidable subproblem exists, called higher-order pattern unification (Dowek et al., 1996). A higher-order pattern is just a simply typed λ-term where every free variable z is applied to a sequence of distinct bound variables. So, $\lambda x.\, z\, x$ is a higher-order pattern but $\lambda x.\, z\, y$ is not.

Full β-reduction is not required to compute the normal forms of higher-order patterns: let β_0-conversion be the restriction of β-conversion to redexes of the form $(\lambda x.\, t)\, x$. The restricted form of higher-order abstract syntax over higher-order patterns and using higher-order pattern unification to decide equality modulo $\alpha\beta_0\eta$-conversion is known as λ-tree syntax. This approach has been used in tools like Abella (Gacek et al., 2009) and Bedwyr (Baelde et al., 2007), respectively an interactive theorem prover and model checker for λ-tree syntax. These systems use the ∇-quantifier (Miller and Tiu, 2005) for building generic proofs about terms with locally-scoped binders.

Finally, a brief word on expressiveness. The nameless representation of object-language bound names in terms of meta-level binders means that some definitions which rely on direct access to the bound name are difficult (maybe even impossible) to encode in HOAS systems. A simple example is the "not a free variable" relation, $x \notin FV(t)$. This can be axiomatised as an inductive definition

$$\frac{x \neq x'}{x \notin FV(x')} \qquad \frac{x \notin FV(t) \qquad x \notin FV(t')}{x \notin FV(t\, t')} \qquad \frac{}{x \notin FV(\lambda x.\, t)} \qquad \frac{x \neq x' \qquad x \notin FV(t)}{x \notin FV(\lambda x'.\, t)}$$

which is the thing that one would first write down (Lakin and Pitts, 2009). The difficulty for a nameless representation is the third rule, which has both bound and free occurrences of the variable x. An extensionally equivalent HOAS representation would probably do without the third rule altogether. In general, it is unclear how to translate a definition from inference rules into a HOAS representation: in many examples a degree of expert knowledge seems necessary.

- **Nominal abstract syntax (Gabbay and Pitts, 2002; Gabbay, 2000)** uses an explicitly named representation of binding. This ties in with the Barendregt variable convention and the use of individual representatives to represent α-equivalence classes. Nominal techniques make it easier to write down (and reason about) binders compared to nameless styles.

  Object-language names are represented by names n (the term "atom" is often used in the literature for historical reasons—we use "name" here for the sake of internal consistency). These are drawn from some countably infinite set *Name*. It is important that there are infinitely many names to choose from, so that there are always new ones available to use. These names follow the permutative convention described above, so distinct names n and n' in the meta-language stand for distinct object-language names.

  Name sorts are declared explicitly and ML-style datatypes for representing abstract syntax are declared in a nominal signature (Urban et al., 2004). For example, the type declarations

for the representing the λ-calculus might look like

```
nametype var
datatype lam = Var of var | Lam of [var]lam | App of lam * lam
```

where the name sort `var` contains meta-level names representing λ-calculus variables. The type `[var]lam` is inhabited by *abstraction* terms of the form <n>t, which represent the binding of a name n whose lexical scope is the term t. For example, the K combinator ($\lambda x.\, \lambda y.\, x$) is encoded as the *nominal term*

```
Lam <n>(Lam <n'>(Var n)).
```

A brief note on terminology: we will use the term "binder" to refer to an actual binding occurrence of a name, i.e. one which is identified up to α-renaming of the bound name. An example of a binder is $\lambda x.\, t$. We say "abstraction" for a data structure like <n>t, which models an object-language binder but is itself not identified up to α-equivalence.

The theory of nominal sets relies on permutative renamings. These have better logical properties than ordinary capture-avoiding renamings, in part because they are bijective. Nominal logic relies on the key property of *equivariance*, that is, closure under name-permutations (see Section 2.5 for more detail). As an example, if we take two α-equivalent nominal terms

```
Lam <n>(App (Var n, Var n'))          Lam <n''>(App (Var n'', Var n'))
```

and apply the name-swapping $(n\, n')$ to t, which involves walking the abstract syntax tree t and replacing every occurrence of n with n' and vice versa (including those in abstraction position), then we get the terms

```
Lam <n'>(App (Var n', Var n))          Lam <n''>(App (Var n'', Var n))
```

which are still α-equivalent. This property clearly does not hold of other kinds of simple non-bijective renaming: we have presented such examples above.

Another key aspect of nominal logic is the *finite support* assumption. A support of an object x is any set $\overline{n} \subseteq \mathit{Name}$ such that $(n\, n') \cdot x = x$ for all $n, n' \notin \overline{n}$. If an object has a finite support, it must have a smallest one, which we call *the support* of x. For our purposes, the support of a term corresponds to its free names—see (Pitts, 2006) for more detail. The support tells us when a name is *fresh for* a term, i.e. does not appear free in it. The freshness relation is written $n \mathrel{\#} t$, and means that n is not in the support of t. This models the "$n \notin FV(t)$" side-condition that is often used to rule out name-capture in definitions.

Finite support is crucial because it ensures (broadly speaking) that we can always find a new name which is fresh for a particular term. This permits a name generation operation. By the permutative convention, the generated names are globally fresh. Nominal logic (Pitts, 2003) gives a first-order characterisation of notions such as α-equivalence, freshness and fresh name quantification. There, the syntax for fresh name generation is $\text{И} n.\ \varphi$, which is read "for a fresh name n, φ holds". In FreshML (Shinwell, 2005) there is a `fresh` keyword, which produces a fresh name when evaluated. The И-quantifier and the ∇-quantifier of HOAS are similar in many ways and may well be related (Gabbay and Cheney, 2004).

A common feature of work in nominal logic are "some/any" properties, such as (Gabbay and Pitts, 2002, Proposition 4.10): for any formula φ and list of distinct names n, $\vec{x}$ (where $\vec{x}$ is the set of free names of φ) the following are all equivalent.

$$\forall n \in \mathit{Name}.\ (n \mathrel{\#} \vec{x} \Longrightarrow \varphi) \qquad \exists n \in \mathit{Name}.\ (n \mathrel{\#} \vec{x} \wedge \varphi) \qquad \text{И} n \in \mathit{Name}.\ \varphi$$

The intuition here is that if the property φ holds for *some* fresh name then it works for *any* fresh name, and vice versa. Indeed, we will prove a some/any property for the semantics of schematic inductive definitions in Section 2.5.3.

- **Locally nameless representation (McKinna and Pollack, 1999; Aydemir et al., 2008)** is a hybrid between the nominal and de Bruijn approaches to abstract syntax. The key idea is that free and bound variables are treated differently: free variables are named explicitly (in the nominal style) whereas bound variables are represented using de Bruijn indices. The λ-term $\lambda x.\, \lambda y.\, x\, z$ might be represented as

$$\lambda\, \lambda\, (\mathtt{BV}\, 2)\, (\mathtt{FV}\, z)$$

  where BV tags the index of a bound variable and FV denotes a free one. The advantage of such a representation is that one gains some of the benefits of nominal abstract syntax while keeping the pleasant property that α-equivalent terms are syntactically identical. However, some of the pain of de Bruijn indices persists. Furthermore, when one moves under a λ-binder, some bound variables may become free and must be substituted away with fresh names. Locally nameless representations have been used to tackle the POPLMARK challenge (Leroy, 2007).

αML takes a nominal approach to handling binders. This allows us to achieve a concrete syntax for the meta-language which is fairly close to that of informal mathematics, while still providing convenient features for encoding languages with binding constructs. As one might expect, αML draws on previous work in this area and addresses some of its shortcomings. Therefore, we now examine some existing nominal meta-programming systems of particular relevance to our discussion.

- **FreshML (Shinwell, 2005)** is a functional programming language which uses the nominal approach to binders and α-conversion. It was first described in (Pitts and Gabbay, 2000) and an implementation in a production compiler (Fresh Objective Caml) was described in (Shinwell, 2005). The advantages of traditional functional programming are that datatypes representing syntax can be defined by induction, and functions can be defined by recursion over those datatypes in a natural way.

  In FreshML, object-language names are represented by names, which are generated freshly when required and written using normal lexical variables x of some name sort. Abstractions are constructed using the standard <*x*>*e* syntax and deconstructed by generative unbinding (Pitts and Shinwell, 2008). This involves generating a fresh name y and swapping all occurrences of x for y throughout the expression e. To quote (Shinwell et al., 2003): "*the essence of FreshML is swapping*". In practice, this unbinding operation is integrated with the operational semantics of standard ML pattern-matching syntax, and its operation is largely hidden from the user. This allows concise definitions: the capture-avoiding substitution function $t[t'/x]$ over λ-terms can be defined as follows.

```
let rec sub t t' x = match t with
  Var y -> if x=y then t' else t
| App (t1,t2) -> App(sub t1 t' x, sub t2 t' x)
| Lam <y>t'' -> Lam(<y>(sub t'' t' x));;
```

  The first and third clauses of the pattern-match are of interest. In the first, a straightforward equality test tells us whether to perform the substitution or not. The third clause uses generative unbinding behind the scenes to ensure that the bound name y from the pattern

is freshened before the body of the clause is evaluated. This prevents name-capture and mirrors informal use of the Barendregt variable convention.

The key correctness result for FreshML is that terms involving binders can be encoded in the meta-language so that contextual equivalence and α-equivalence coincide. This was proved both denotationally (Shinwell and Pitts, 2005) and operationally (Pitts and Shinwell, 2008).

FreshML is an impure language because the generation of names can be observed as a side-effect. The earlier FreshML-2000 language (Pitts and Gabbay, 2000) had a freshness inference system for statically rejecting programs where freshly generated names were returned un-abstracted (and hence were observable). This was dropped from the version of FreshML described in (Shinwell et al., 2003) because it rejected too many reasonable-looking programs. The downside is that one can write meaningless programs—see (Shinwell, 2005, Section 6.8.2) for examples.

Recent work by Pottier (Pottier, 2007) describes a tractable and practical decision procedure for rejecting impure programs in a FreshML-like language, using user-supplied freshness assertions. That paper employs a system of binding specifications which is richer than our single name-binding operator $<x>t$. These originate from work on Cαml (Pottier, 2006), a tool which auto-generates Objective Caml code from such a binding specification.

- **Nominal unification (Urban et al., 2004):** extends first-order unification to work on terms modulo α-conversion. Problems consist of equality ($t = t'$) and freshness ($n \# t'$) constraints, which are solved in the context of a *freshness environment* ∇ of freshness assumptions $n \# X$ between a name and a meta-variable. Such assumptions are needed to constrain the free names in an unknown term, so if $(n \# X) \in \nabla$ then X cannot be replaced by any term which has a free occurrence of n.

  The results of nominal unification are a new freshness environment and a substitution of terms for meta-variables. The solution is unique and most general (Urban et al., 2004, Theorem 3.7), and an implementation using graph-based data structures for maximal sharing has been shown to have polynomial time complexity (Calves and Fernández, 2007). First-order unification is decidable in linear time (Paterson and Wegman, 1976), but it is an open question whether nominal unification is too.

- α**Prolog (Cheney, 2004b)** is a logic programming language which uses nominal logic to compute over syntax modulo α-conversion. It was introduced in (Cheney and Urban, 2004), and model-theoretic, proof-theoretic and operational semantics for the language are developed in (Cheney, 2004b) and (Cheney and Urban, 2008).

  αProlog programs use explicit, fresh names like FreshML, and a similar abstraction term-former. The following example, paraphrased from (Cheney, 2004b, Example 2.2.1), presents αProlog code for type-checking λ-terms, expressed as a three-place relation `typ`. The type `ctx` is for type environments implemented as lists of pairs consisting of a variable and its type, and the types `exp` and `ty` represent object-language expressions and types respectively.

```
pred typ(ctx,exp,ty).
typ(G, Var(X), T)            :- mem((X,T) ,G).
typ(G, App(M,N), T')         :- typ(G, M, Fun(T,T')), typ(G, N, T).
typ(G, Lam(<x>M), Fun(T,T')) :- x#G, typ((x,T')::G, M, T').
```

  The three clauses in this program correspond to the three typing rules for λ-calculus from Figure 1.1. The left-hand side of each is the head of the clause, which corresponds to the

conclusion of the inference rule. The first clause uses a predicate `mem`, which tests list membership and is used to check whether the appropriate typing assumption exists in the environment `G`. The third clause exploits the nominal meta-programming capabilities of αProlog: the freshness constraint `x # G` ensures that the bound name `x` is not free in the domain of the typing environment, as is standard. The typing judgement can then move underneath the λ-binder. An example query to find a type for the identity function in the empty typing environment would be

```
typ([],Lam(<x>(Var(x))),T).                                  (1.5)
```

Logic programming is clearly a natural fit for programs which perform proof-search computations over rule-based definitions, because search and backtracking facilities are built into the operational semantics of the language. The system attempts to match the user's query against the heads of the program clauses in turn. If a match is successful, the system attempts to use that rule to construct a derivation of the associated goals, in a process called resolution. For example, the example query (1.5) will fail to match against the first two program clauses before matching against the third.

In Prolog, the resolution process uses first-order (syntactic) unification, and before unification is attempted the program clause is freshened by choosing new meta-variables to prevent name-clashes. However, αProlog must use a more powerful unification algorithm, which takes binders and α-equivalence into account. Continuing the example (1.5), after the query has matched against the third program clause the system must prove the new goal `typ((z,T')::[],Var(z),T'')`, where `z` is a fresh name and `T''` is a new meta-variable.

Nominal unification would seem a natural choice for resolution in αProlog (Cheney and Urban, 2004), because it is efficient and relatively simple to implement. However, there is a theoretical problem: *resolution with nominal unification is incomplete for nominal logic*.

The problem stems from the equivariance property of nominal logic: if a formula φ is valid then all possible permutative renamings $\pi \cdot \varphi$ of that formula must also be valid. One can write αProlog programs which do not compute all solutions to a query because resolution with nominal unification does not work up to an arbitrary permutation. A simple example from (Cheney, 2004b, Section 2.1) consists of the single program clause

```
p(a).                                                        (1.6)
```

where `a` is a name. If we try to solve the goal `p(a)` (which clearly is a consequence of the program in the theory of nominal logic) the system will fail. The program clause is freshened to `p(a')`, causing nominal unification to fail because $\texttt{a} \neq \texttt{a'}$.

Several solutions have been proposed: one can impose a syntactic restriction on αProlog programs to reject those which might be incomplete (Urban and Cheney, 2005; Cheney and Urban, 2008). This works in many cases, but some programs must be rewritten before the system will accept them and the decision procedure is a conservative approximation. Alternatively, one can replace nominal unification with the more powerful *equivariant unification* algorithm (Cheney, 2005a) which unifies terms up to a permutation. This gives the extra power for a complete search procedure for problematic examples such as (1.6) but is complicated and computationally expensive. Equivariant unification will be discussed further in Section 1.4.2 below.

The remainder of this section comprises more specific examples of tools for language design and meta-programming which are related to αML. Some of these also employ nominal techniques. By necessity the list is brief—other relevant work is discussed and cited at appropriate points in the text.

- **Functional logic programming languages:** the most prominent of these multi-paradigm languages is Curry (Hanus, 1997); another is Mercury (Somogyi et al., 1996). From our perspective, the major drawback of most existing functional logic languages is that they lack built-in support for programming with names and binders. A notable exception is Qu-Prolog (Nickolas and Robinson, 1996) which extends Prolog with features for quantifying object-language variables and explicit substitutions. The Qu-Prolog unification algorithm is semi-decidable, and is closer in spirit to equivariant unification than to nominal unification.

  The main problem encountered by functional logic languages is how to deal with the application of a function (which are typically defined by cases) to an unknown term, i.e. expressions like `f (X)`. There are two common solutions:

  - *residuation* involves suspending the current computation in the hope that some other thread may compute an instantiation for `X`. The language must include concurrency primitives and the strategy is not guaranteed to succeed. If no instantiation is found, the computation fails by *floundering*. The operational semantics of residuation and concurrency in Curry are discussed in (Albert et al., 2002).
  - *narrowing* amounts to trying all possible (outermost) term constructors, in the hope that one or more of these "guesses" may succeed. A strategy for performing narrowing only when absolutely necessary is described in (Antoy et al., 2000).

  Consider the following example, taken from (Tolmach et al., 2004). A datatype is defined (using Curry syntax) for colours and gives a rule-based definition for a (partial) function `mix` that combines certain primary colours to produce new colours.

```
data Color = Red | Yellow | Blue | Orange | Violet | Green

mix Red Blue = Violet
mix Yellow Blue = Green
mix Yellow Red = Orange

a1,a2 :: Color
a1 = mix Red Blue
a2 = mix Yellow x where x free
```

  Evaluation of `a1` proceeds deterministically because both arguments to `mix` are fully instantiated. In `a2`, however, the variable `x` is declared as a logic variable which means that the call to `mix` cannot happen without an instantiation for `x`. According to the rules, there are only two possibilities and the operational semantics non-deterministically tries both of them, producing the two answers `Green` and `Orange`. The weakness of the narrowing strategy is that it cannot handle primitive functions such as the infixed addition operator `(+)`, which are not defined by a set of rules. The expression "$3 + \texttt{x}$" (where `x` is a logic variable) cannot be reduced using narrowing, as there are (theoretically) infinitely many ways to instantiate the logic variable. Curry handles expressions such as this using a combination of residuation and concurrent execution (Hanus, 2007, Section 2.4). The thread trying to evaluate "$3 + \texttt{x}$" is suspended (residuated) and waits in the hope that the evaluation of some other concurrently-executing thread will instantiate `x` sufficiently to allow the addition to be evaluated deterministically. This may never happen, in which case the computation flounders.

  In practice, Curry supports both residuation and narrowing and the programmer can decide which to use in any given situation. αML only supports narrowing so as not to overcomplicate the language.

- **Nominal Isabelle (Urban, 2008)** is actually a package for the Isabelle/HOL theorem prover (Nipkow et al., 2002) which adds the underlying theory of nominal sets and provides features for working with nominal datatypes that involve names and binders. The system can semi-automatically derive strong induction principles (Pitts, 2006; Urban et al., 2007) for proving properties of languages involving binders such that the proof really does work up to α-renaming. Other work includes the derivation of inversion principles which permit both top-down and bottom-up reasoning over inductive definitions (Berghofer and Urban, 2008). The nominal datatype package has been used in numerous formalisations such as those described in (Urban et al., 2008; Urban and Nipkow, 2008).

- `ott` **(Sewell et al., 2007)** allows the syntax and semantics of a language to be defined in a centralised location, which simplifies the management and modification of the language definition. User-defined syntax can be used in the definition of the semantic rules, so they look convincingly like their pen-and-paper equivalents. For example, the record typing rule can be written in `ott` as follows.

```
 G |- t1:T1 ... G |- tn:Tn
--------------------------------------------
 G |- {l1=t1,...,ln=tn} : {l1:T1,...,ln:Tn}
```

  The system can automatically generate datatype definitions and boilerplate code for various theorem provers (including Isabelle, Coq and HOL), along with LaTeX code for typesetting the ASCII definitions. It has already been used on some fairly large formalisations such as the operational semantics of $\text{OCaml}_{\text{light}}$ (Owens, 2008).

  The `ott` meta-language includes a rich language of binding specifications, which allow complicated binding structures like structured patterns and mutually-recursive `let` bindings to be expressed. However, the current implementation of the tool produces code for fully concrete representations, i.e. not quotiented by α-equivalence. An experimental extension to the Coq backend generates code for a locally-nameless representation of syntax involving binders.

- **Nominal techniques in Scheme:** particularly relevant is the αKanren system (Byrd and Friedman, 2007), which embeds nominal unification and its associated proof-search procedure over nominal terms within Scheme. This has been used to build a concise theorem prover for first-order logic, with the nominal subsystem taking care of binding issues (Near et al., 2008).

  It is interesting to note that the generative unbinding strategy used in FreshML (Pitts and Shinwell, 2008), one of the key ideas of nominal meta-programming, seems to have been independently invented in the late 1980s in the context of hygiene preservation during Scheme macro expansion (Kohlbecker et al., 1986). Furthermore, a later paper (Dybvig et al., 1992) addressed efficiency concerns with a lazy propagation of renamings remarkably similar to that proposed in (Shinwell, 2005, Section 7.1.1).

- **PLT Redex (Matthews et al., 2004)** was designed specifically for animating small-step reductions from a description of a language semantics. Languages are specified using Scheme-like syntax and there is no support for dealing with object-language binders up to α-equivalence. Furthermore, the animation is restricted to applying a single rewrite rule to a single term-in-context at each step. However, PLT Redex does include a powerful graphical visualisation toolkit.

1.4 The novelty of αML

αML follows in the tradition of higher-order typed functional programming languages, and in particular of FreshML (Shinwell, 2005). It also incorporates some aspects of the logic programming paradigm, drawing on αProlog (Cheney and Urban, 2004). The novel features of αML can be summarised as follows.

- **Existential variable generation.**
The syntax $\exists x : E.\, e$ creates a new meta-variable of type E, whose scope is the expression e. The meta-variable x stands for an unknown object-language *term* (there are restrictions on the type E which rule out function types, for example). In particular, a meta-variable of a name sort N represents an object-language *name* which may or may not be distinct from other object-language names.

- **Non-deterministic branching.**
The expression $e \parallel e'$ causes a non-deterministic branch which explores both e and e' in some fair way. This allows proof-search computations (in which certain search avenues may fail and backtrack) to be expressed elegantly. The operational semantics does not specify any particular search strategy or treatment of backtracking—we defer such practical issues to our discussion of the implementation in Chapter 7.

- **Abstraction syntax.**
Following most nominal meta-programming languages, we write $\langle x \rangle e$ to stand for the binding of the name x in the expression e, where any value produced by evaluating e should represent an object-language term. This is *not* a binder in the meta-language.

- **Equality constraints.**
The constraint $e = e'$ checks whether there exists some instantiation of the meta-variables within the terms (with α-equivalence classes) which makes them α-equivalent. Having a built-in α-equivalence test allows users to check their terms for equality, safe in the knowledge that the structural congruence of α-equivalence is being respected. On a practical note, equality constraints are needed to pattern-match against the conclusion of inference rules during proof-search.

- **Freshness constraints.**
Like the freshness constraints from nominal unification, these take the form $x \,\#\, e$ and model the common "not free in" relation between a name x and the object-language term represented by e. When the term on the right-hand side of the # is of a name sort, then the constraint corresponds to a test for inequality between two names.

When added to an eager core ML-like language, these features give a minimalist calculus for animating rule-based inductive definitions involving binders up to α-equivalence. Other kinds of computation, for example heavily numerical algorithms, would be difficult to implement in αML. We will defer the presentation of full αML programs until the language of inductive definitions and the αML meta-language itself have been formally introduced. For readers wishing to skip ahead, an example is presented in Appendix D.

1.4.1 Functional logic programming

The melding of functional and logic programming features in αML is somewhat different from that of traditional functional logic languages such as Curry (Hanus, 1997). There, the emphasis tends to be on integrating functions into the logic programming paradigm. In contrast, the

design of αML integrates elements of logic programming into a functional programming language. This enabled us to apply some well-established techniques from work on functional programming languages, such as on program equivalence (Howe, 1996; Pitts, 2005).

The multi-paradigm approach seems beneficial for encoding operational semantics and type systems. Logic programming is compelling for implementing the search procedure over inductive definitions, but some aspects of those definitions are more naturally defined in a functional style. For example, the capture-avoiding substitution notation $t'[t/x]$ tends to be viewed as denoting a function

$$\texttt{sub} : \texttt{term} * \texttt{term} * \texttt{var} \rightarrow \texttt{term}$$

which takes a term, a variable and a term and produces another term. It is clearly possible to define capture-avoiding substitution in an equivalent relational style, as

$$\texttt{sub} \subseteq \texttt{term} * \texttt{term} * \texttt{var} * \texttt{term},$$

where sub (t, t', x, t'') is interpreted as $t[t'/x] =_\alpha t''$. However, it seems sensible to do things in a similar way to informal practice, and the combination of functional and logic programming constructs allows us to express the standard β-rule as

```
     (sub t' x t) = t''
------------------------------ [reduce_beta where x:var, t,t',t'':lam]
 REDUCE(App((Lam<x>t),t'),t'')
```

where sub (t', x, t) returns the term $t'[t/x]$. This is real αML code—see Appendix D for more examples. We have found the rule-based syntax to be convenient in practice, as it allows near-verbatim transcription of inference rules from papers into the programming language.

1.4.2 Representation of names

The representation of names in αML merits further discussion. Suppose that N is a type of object-language names, and consider the following αML expression.

$$\exists x : N. \exists y : N. e$$

Evaluating this expression generates two meta-variables (x and y) which range over object-language names, then proceeds to evaluate e. The key point: *just after y is generated, the system is agnostic to the relationship between x and y.*

Both x and y range over the entire set of object-language names, so they could both refer to the same name or to different ones. This removes the permutative convention, as syntactically distinct meta-variables could be *aliased* to refer to the same object-language name. For example, the αML value

$$\texttt{Lam}\,\texttt{<}x\texttt{>}(\texttt{Lam}\,\texttt{<}y\texttt{>}(\texttt{Var}\; x))$$

could represent either or both of the distinct α-equivalence classes

$$(i) \quad [\lambda n. \lambda n'. n]_\alpha \qquad\qquad (ii) \quad [\lambda n. \lambda n. n]_\alpha$$

depending on whether we assert that $x \neq y$ (just *(i)*), $x = y$ (just *(ii)*), or leave it unspecified (both *(i)* and *(ii)*), assuming that $n \neq n'$.

The representation of abstract syntax in αML generalises traditional nominal techniques which use permutative names to represent object-language names. Permutative behaviour can

be modelled in αML by adding appropriate freshness constraints between names. The above example can be made to follow the permutative convention by modifying it to

$$\exists x : N.\, \exists y : N.\, (x \# y)\ \&\ e$$

where e_1 & e_2 evaluates e_1 and e_2 sequentially. In the modified example, evaluation of e proceeds under the constraint that x and y refer to distinct names.

Our motivation for removing the permutative convention is to produce a simple search procedure which is complete for our language of inductive definitions. As mentioned above, to achieve completeness in αProlog one can use equivariant unification for resolution. This solves the problem, but at a cost: equivariant unification generalises the term language of nominal unification significantly, for example there are variables ranging over unknown permutations and nested permutations are permitted. In the world of equivariant unification we can write terms like

$$<(((QQ')\, N)\, (Q^{-1}\, N))\, n>(Q'\, n)$$

even though the meaning of such a term is by no means obvious. In the above example, Q and Q' are unknown permutations, N is an unknown name (like our variables of name sort which may be aliased) and n is a permutative name. These extra features means that implementing the equivariant unification algorithm is somewhat complicated. Furthermore, the algorithm is computationally expensive, in fact NP-complete (Cheney, 2004a).

We propose a simpler system: removing the permutative convention and doing away with permutative names altogether gives sufficient power to overcome the incompleteness issues of proof-search powered by nominal unification. Furthermore, our constraint transformation algorithm (described in Chapter 6) is relatively easy to understand and to implement. Our constraint problem is NP-complete (see Section 3.4.1 for a simple proof), and it follows that *it is equivalent to equivariant unification*. We believe that the simplicity of our approach gives it significant practical advantages over approaches based on equivariant unification.

1.5 Dissertation overview

The contributions of this dissertation can be summarised as follows.

- Complete proof search over inductive definitions involving explicitly-named binders currently requires equivariant unification, which is very complicated to implement and NP-complete. We show how a generalisation of nominal terms, which we will refer to as *non-permutative* nominal terms, lets us achieve completeness in a far simpler way.

- We demonstrate this approach through its application in the design and implementation of the αML meta-language for prototyping systems defined as inductive rules.

The dissertation structure is as follows. The following chapter formalises general notions of schematic formulae and schematic inductive definitions involving binders and equips these definitions with a model-theoretic semantics based on α-equivalence classes of abstract syntax trees. In Chapter 3 we introduce the αML meta-language and present its operational semantics in the form of a non-deterministic small-step transition relation. Chapter 4 describes a straightforward encoding of inductive definitions into αML and proves soundness and completeness results with respect to the semantics of inductive definitions. In Chapter 5 we develop a notion of operational equivalence for αML programs. We then relate standard notions of equivalence between individual abstract syntax trees (α-equivalence) and formulae (equality of denotations) to operational equivalence of their encodings in αML. We also discuss a finer

equivalence relation and issues pertaining to fresh name generation. In Chapter 6 we describe an algorithm which can solve the underlying constraint problem and in Chapter 7 we outline the implementation of the αML language and of the constraint transformation algorithm. Finally, in Chapter 8 we conclude and discuss some possible directions for future work.

Further details of certain proofs and in-depth discussions of certain aspects of the implementation are presented as Appendices.

1.6 On collaboration

The process of language design, and research in general, tends to be a collaborative endeavour. The work described in this dissertation is no exception, and we finish this chapter with a brief discussion of the sections which are the result of collaborative work.

The text of this dissertation was written entirely by the author. The content of Chapter 2 and Chapter 3 was developed by the author in collaboration with Andrew Pitts, and the content of the remaining chapters is entirely the author's own work. The implementation of αML described in Chapter 7 and Appendix C builds upon code by Andrej Bauer (Bauer, 2008) and depends on some external libraries, whose authorship is acknowledged in the text. Otherwise, the implementation of αML is due to the author.

Chapter 2

α-inductive definitions

"Everything is vague to a degree you do not realize till you have tried to make it precise."

—B. Russell

In this chapter we develop a class of inductively defined relations between α-equivalence classes of terms, which we refer to as *α-inductive definitions*. We deal with α-equivalence classes because the terms may contain binding constructs. These correspond to the abstract syntax trees of programs modulo α-equivalence. The forms of the abstract syntax trees are specified using nominal signatures (Urban et al., 2004). The rules that we deal with will be schematic, in that they constitute a template for creating rule instances, for some means of instantiating variables with α-equivalence classes of ground terms.

This is an important first step because schematic inductive definitions are ubiquitous and are therefore a worthy object of study in their own right. We present a syntactic class of formulae over α-equivalence classes, and combine these to produce full inductive definitions of relations. We then define satisfaction of formulae and give a simple model-theoretic semantics to α-inductive definitions.

2.1 Signatures and equality types

Our first ingredient is some way of specifying the abstract syntax of the language that we wish to model. To this end we use nominal signatures (Urban et al., 2004), which extend first-order algebraic signatures with constructors that bind names in a given scope.

Definition 2.1.1 (Nominal signatures). A nominal signature Σ consists of:

- a finite set $\mathbb{N}_\Sigma$ of *name sorts*, ranged over by N;
- a finite set $\mathbb{S}_\Sigma$ of *nominal data sorts*, disjoint from $\mathbb{N}_\Sigma$ and ranged over by S; and
- a finite set $\mathbb{C}_\Sigma$ of *constructors* $K : E \to S$, where the argument type E is an *equality type* of Σ, generated by the grammar:

$$
\begin{array}{rcll}
E \in Ety_\Sigma & ::= & S & \text{(nominal data sorts)} \\
 & & N & \text{(name sorts)} \\
 & & [N]\,E & \text{(name abstractions)} \\
 & & E * \cdots * E & \text{(tuples)} \\
 & & \mathtt{unit} & \text{(unit).}
\end{array}
$$

◇

The non-standard elements here are the the name sorts N, which are inhabited by object-language names, and name abstraction sorts $[N]E$, which are inhabited by object-language terms where a single name of sort N is bound in a term of type E. We refer to these as "abstractions" because they are not treated as binders at the meta-level but are simply used to model object-language binders. This follows the type system of FreshML (Shinwell et al., 2003). *Unless specified otherwise, we will henceforth assume the existence of a nominal signature* Σ.

At first glance, the ability to bind a single name within a term seems quite restrictive compared to more expressive schemes such as those of `ott` (Sewell et al., 2007) and Cαml (Pottier, 2006). This design decision was made for simplicity and also because one can get quite a long way using just single binding. There are currently no concrete theorems on the expressive power of the various binding specification languages.

We use the term *equality type* in the sense of Standard ML (Milner et al., 1997, Section 4.4), to mean those types whose values admit a decidable notion of equality. In particular, function types are not equality types. This will allow us to reject programs at compile-time if they involve equality constraints that could only be solved using higher-order unification techniques or if they attempt to create existential variables to stand for higher-order terms.

By way of an example, Figure 2.1 defines a nominal signature $\mathcal{F}$ for the polymorphic λ-calculus, also known as System F (Girard, 1993), which will serve as a running example throughout this dissertation. A grammar for System F types τ and terms M is as follows.

$$
\begin{array}{rcll}
\tau & ::= & \alpha & \text{(type variable)} \\
 & & \tau \to \tau & \text{(function type)} \\
 & & \forall \alpha.\, \tau & \text{($\forall$-type).} \\
M & ::= & x & \text{(variable)} \\
 & & \lambda x : \tau.\, M & \text{(λ-abstraction)} \\
 & & M\, M & \text{(application)} \\
 & & \Lambda \alpha.\, M & \text{(type generalisation)} \\
 & & M\, \tau & \text{(type specialisation).}
\end{array}
$$

It is instructive to consider System F because of its variety of binding forms. There are two sorts of name that may be bound: type variables α and term variables x, which are modelled by the name sorts `tyvar` and `var`. There are also two sorts of data: types and terms which correspond to the nominal data sorts `type` and `term` respectively. Finally, there are three different flavours of binding presaent:

- binding of a type variable α in a type τ by the universal quantification operator $\forall \alpha.\ \tau$;
- binding of a term variable x in a term M in the λ-abstraction $\lambda x : \tau.\, M$; and
- binding of a type variable α in a term M to produce the type generalisation term $\Lambda \alpha.\, M$.

These are modelled by the `ForAll`, `Lam` and `Gen` constructors respectively, whose argument types reflect the kinds of name that they bind. The signature prevents us from binding a type variable where a term variable is expected, for example. Figure 2.1 also includes a nominal data sort `tenv` for encoding type environments as lists using the `Nil` and `Cons` constructors. The members of these lists have type `var * type`, and represent a term variable associated with its type. We will use this later when we define the System F typing judgement.

The presence of type annotations in System F terms makes type-checking trivial—without them, it would be undecidable (Wells, 1994). It is worth noting that an alternative argument type for the `Lam` constructor could be

$$\texttt{Lam} : (\texttt{type} * [\texttt{var}]\texttt{term}) \to \texttt{term}$$

Name sorts	Data sorts	Constructors
`tyvar`	`type`	`TyVar : tyvar → type`
`var`		`Fun : type * type → type`
		`ForAll : [tyvar]type → type`
	`term`	`Var : var → term`
		`Lam : [var](type * term) → term`
		`App : term * term → term`
		`Gen : [tyvar]term → term`
		`Spec : term * type → term`
	`tenv`	`Nil : unit → tenv`
		`Cons : (var * type) * tenv → tenv`

Figure 2.1: Example nominal signature $\mathcal{F}$ for System F

in other words we may choose to have the type annotation either inside or outside the scope of the bound variable. We have this choice because the bound name is of sort `var`, and variables of this sort can never appear in types (as the argument types of `TyVar`, `Fun` and `ForAll` do not mention `var` or any type involving `var`).

2.2 Ground trees and α-equivalence classes

The universe of discourse for providing a semantics for inductive definitions is typically α-equivalence classes of ground terms. An inductively-defined relation will then carve out a subset of these—in order to formalise inductive definitions we first need to define a suitable set of ground terms.

In keeping with the nominal approach to abstract syntax (Gabbay and Pitts, 2002) bindable names are represented explicitly in the syntax of ground terms. We fix a countably infinite set *Name* of names to stand for object-language names which may be bound. The meta-variable n ranges permutatively over these. We assume the existence of a total function *sort* which maps every name n to a name sort $N \in \mathbb{N}_\Sigma$ and is such that there are infinitely many names assigned to every name sort. We say that $n \in \mathit{Name}(N)$ if $\mathit{sort}(n) = N$.

Definition 2.2.1 (Ground trees). We write Tree_Σ for the set of all syntax trees over the nominal signature Σ. With names (and unit) as our building blocks, we define classes $g \in \mathit{Tree}_\Sigma(E)$ of syntax trees of the various equality types by constructor application, tupling and name abstraction, as follows.

$$\frac{\mathit{sort}(n) = N}{n \in \mathit{Tree}_\Sigma(N)} \qquad \frac{}{() \in \mathit{Tree}_\Sigma(\mathtt{unit})} \qquad \frac{g_1 \in \mathit{Tree}_\Sigma(E_1) \quad \cdots \quad g_n \in \mathit{Tree}_\Sigma(E_n)}{(g_1, \ldots, g_n) \in \mathit{Tree}_\Sigma(E_1 * \cdots * E_n)}$$

$$\frac{g \in \mathit{Tree}_\Sigma(E) \qquad (K : E \to S) \in \Sigma}{K\, g \in \mathit{Tree}_\Sigma(S)} \qquad \frac{\mathit{sort}(n) = N \qquad g \in \mathit{Tree}_\Sigma(E)}{\mathtt{<}n\mathtt{>}g \in \mathit{Tree}_\Sigma(\mathtt{[}N\mathtt{]}\,E)}$$

◇

This definition is fairly standard—the most interesting rule is for abstractions, which enforces syntactically that we can only bind a single name, which (by the rules) must be of a

name sort N. Our ground trees correspond precisely to the ground nominal terms of (Urban et al., 2004)—i.e. those which do not contain logic variables.

The astute reader will note that we have not yet made any mention of α-equivalence with regard to our abstract syntax trees. This means that if $n \neq n'$ then the ground trees $<n>n$ and $<n'>n'$ would be regarded as distinct trees. Clearly this is not desirable as abstract syntax is typically identified up to α-conversion. Therefore, we must define a notion of α-equivalence on ground trees.

Definition 2.2.2 (α-equivalence for ground trees). We write $g =_\alpha g' : E$ for the congruence relation induced on pairs of ground trees (of the same equality type) by considering the name abstraction term-former $<n>g$ as a binder. ◇

This definition of α-equivalence paraphrases that of (Barendregt, 1984). Alternative definitions exist, for example (Urban et al., 2004, Figure 2) has a definition of α-equivalence for ground trees phrased in terms of name-swappings. We do without name swappings here because they do not feature in our term language, so it is preferable to define α-equivalence without them.

Definition 2.2.3 (α-trees). Let $\alpha\text{-}\mathit{Tree}_\Sigma(E)$ be the set of all $=_\alpha$-equivalence classes of ground trees of type E, which we call *α-trees*. We let t range over α-trees. If $g \in \mathit{Tree}_\Sigma(E)$ then we write $[g]_\alpha$ for the set $\{g' \mid g =_\alpha g' : E\}$ of all ground trees which are α-equivalent to g. If $g \in \mathit{Tree}_\Sigma(E)$ then $[g]_\alpha \in \alpha\text{-}\mathit{Tree}_\Sigma(E)$. ◇

Lemma 2.2.4. *If $t \in \alpha\text{-}\mathit{Tree}_\Sigma(N)$ then $t = [n]_\alpha = \{n\}$ for some $n \in \mathit{Name}(N)$.*

Proof. By inspection of the rules from Definition 2.2.1, using the fact that constructors cannot produce trees of name sort (because the sets of name sorts N and nominal data sorts S are disjoint). □

α-trees will form the basis for the semantics of our language of inductive definitions. These represent the object-language terms quotiented by α-equivalence which are so frequently used in informal mathmatical parlance. Over the next few sections we will introduce schematic patterns and rules, and illustrate their semantics in terms of α-trees. First, though, we present standard notions of the free names of a ground term and of freshness ("not a free name of").

Definition 2.2.5 (Free names and freshness). Suppose that $t \in \alpha\text{-}\mathit{Tree}_\Sigma(E)$. Then, we write $FN(t)$ for the finite set of names which occur free in some/any[1] ground tree $g \in \mathit{Tree}_\Sigma(E)$ such that $t = [g]_\alpha$. Hence, $n \in FN([g]_\alpha)$ if and only if n occurs in g and at least one of the occurrences is not within the scope of a $<n>(-)$ name abstraction.

Furthermore, if $t \in \alpha\text{-}\mathit{Tree}_\Sigma(N)$ and $t' \in \alpha\text{-}\mathit{Tree}_\Sigma(E)$ then (by Lemma 2.2.4) we know that $t = [n]_\alpha$ for some $n \in \mathit{Name}(N)$. Then, we write $t \mathrel{\#} t'$ and say "t is fresh for t'" if and only if $n \notin FN(t')$. ◇

2.3 Syntax of α-inductive definitions

We now move on to the defining the syntax of schematic formulae, inductive rules and fully-fledged α-inductive definitions.

[1] Some/any properties are characteristic of nominal techniques for representing abstract syntax—see (Pitts, 2006) for a rigorous mathematical treatment.

$$\frac{x \in dom(\Delta) \qquad \Delta(x) = E}{\Delta \vdash x : E} \qquad \frac{\Delta \vdash p : E \qquad (K : E \to S) \in \Sigma}{\Delta \vdash K\,p : S} \qquad \frac{}{\Delta \vdash \texttt{()} : \texttt{unit}}$$

$$\frac{\Delta \vdash p_1 : E_1 \qquad \cdots \qquad \Delta \vdash p_n : E_n}{\Delta \vdash (p_1, \ldots, p_n) : E_1 * \cdots * E_n} \qquad \frac{\Delta \vdash x : N \qquad \Delta \vdash p : E}{\Delta \vdash \texttt{<}x\texttt{>}p : [N]\,E}$$

Figure 2.2: Typing rules for schematic patterns

2.3.1 Schematic patterns

Patterns are used in informal mathematics as templates which may be used to produce a (potentially infinite) set of ground instances. To permit this, they contain variables which are instantiated with (α-equivalence classes of) ground terms following some particular instantiation rules. A simple language of patterns will be used to build up schematic formulae and inference rules.

We use a fixed, countably infinite set *Var* of variables as placeholders for unknown α-equivalence classes. We will use various meta-variables, typically x, y, etc., to range over these. Along with the empty tuple, written `()`, these are the basic building blocks of our schematic patterns.

Definition 2.3.1 (Schematic patterns). The set Pat_Σ of *schematic patterns* over a nominal signature Σ is defined by the following grammar.

$p \in Pat_\Sigma$	$::=$	x	(variable)
		$()$	(unit)
		$(p, \ldots, p)$	(tuple)
		$K\,p$	(constructor application)
		$\texttt{<}x\texttt{>}p$	(name abstraction). ◇

In order to assign types to schematic patterns we must first provide types for all the variables contained therein—we write $vars(p)$ for the set of all variables occurring in a pattern p. We also write Δ for an environment which assigns equality types to finitely many variables—$dom(\Delta)$ stands for the set of all variables in the domain of definition of Δ. Figure 2.2 provides rules which define a typing judgement $\Delta \vdash p : E$.

Lemma 2.3.2. *If $\Delta \vdash p : N$ then $p = x$ for some variable such that $x \in dom(\Delta)$ and $\Delta(x) = N$.* □

Recalling the nominal signature $\mathcal{F}$ from Figure 2.1, the System F polymorphic identity function $\Lambda\alpha.\,\lambda x : \alpha.\,x$ could be encoded using the schematic pattern

$$\texttt{Gen}\,\texttt{<}a\texttt{>}\texttt{Lam}\,\texttt{<}x\texttt{>}\texttt{(TyVar}\,a\texttt{,}\ \texttt{Var}\,x\texttt{)}$$

for which we could demonstrate the meta-level typing judgement

$$\{a : \texttt{tyvar}, x : \texttt{var}\} \vdash \texttt{Gen}\,\texttt{<}a\texttt{>}\texttt{Lam}\,\texttt{<}x\texttt{>}\texttt{(TyVar}\,a\texttt{,}\ \texttt{Var}\,x\texttt{)} : \texttt{term}.$$

We now describe the instantiation of schematic patterns, which produces specific ground instances.

Definition 2.3.3 (α-tree valuations). An *α-tree valuation* V is a finite partial function which maps variables to α-trees. We write $dom(V)$ for the domain of the partial function. Given a type environment Δ we write $\alpha\text{-}Tree_{\Sigma}(\Delta)$ for the set of all α-tree valuations V such that $dom(V) = dom(\Delta)$ and $V(x) \in \alpha\text{-}Tree_{\Sigma}(\Delta(x))$ for all $x \in dom(V)$. This ensures that the valuation respects types. ◇

The following lemma formalises the fact that there exists a pattern instantiation operation $[\![p]\!]_V$ which respects both types and α-equivalence classes.

Lemma 2.3.4. *If $\Delta \vdash p : E$, then there is a function $V \in \alpha\text{-}Tree_{\Sigma}(\Delta) \mapsto [\![p]\!]_V \in \alpha\text{-}Tree_{\Sigma}(E)$ where*

$$\begin{aligned} & [\![x]\!]_V = V(x) \\ [\![p]\!]_V = [g]_\alpha \implies & [\![K\,p]\!]_V = [K\,g]_\alpha \\ & [\![()]\!]_V = [()]_\alpha \\ [\![p_1]\!]_V = [g_1]_\alpha \wedge \cdots \wedge [\![p_n]\!]_V = [g_n]_\alpha \implies & [\![(p_1, \ldots, p_n)]\!]_V = [(g_1, \ldots, g_n)]_\alpha \\ V(x) = [n]_\alpha \wedge [\![p]\!]_V = [g]_\alpha \implies & [\![\texttt{<}x\texttt{>}p]\!]_V = [\texttt{<}n\texttt{>}g]_\alpha . \end{aligned}$$

Proof. Using the techniques developed in (Pitts, 2006). □

Remark 2.3.5 (Important points regarding patterns).

1. *Variables x stand for unknown α-trees, not unknown trees.* Hence, a pattern $p \in \alpha\text{-}Tree_{\Sigma}(E)$ describes an α-tree as opposed to a tree—precisely which one depends on how its variables are instantiated by a valuation. This reflects the common practice of leaving α-equivalence implicit and using representatives to stand for the whole class (Barendregt, 1984, Convention 2.1.13). The variable convention is commonly used in informal reasoning to rename bound names to avoid one another, as we do in Section 3.2 where we "*identify expressions up to α-conversion of bound variables*".

2. *Variables may occur multiple times in patterns.* We do not impose any kind of linearity constraint on the occurrences of variables in patterns.

3. *The abstraction term-former is not a binder.* This means that there are no meta-level binding constructs in patterns. In particular, the variable x is considered free in the pattern $\texttt{<}x\texttt{>}p$.

4. *Names do not occur in patterns.* Despite the fact that the meta-language allows us to explicitly name object-level binding occurrences using the $\texttt{<}x\texttt{>}p$ syntax, we use variables x of name sort to perform this reference, as opposed to the underlying concrete names themselves. This fits with informal practice as outlined in Section 1.2: Barendregt uses schematic variables ranging over names throughout Barendregt (1984).

5. *Applying a valuation to a pattern is a "possibly-capturing" form of substitution.* Once again, this reflects common practice when instantiating the meta-variables in schematic rules. This is another reason why we do not identify patterns up to α-renaming of abstracted variables. For example, given distinct variables x, y, z we cannot regard the patterns $\texttt{<}x\texttt{>}z$ and $\texttt{<}y\texttt{>}z$ as equivalent because the valuation $V = \{x \mapsto [n]_\alpha, y \mapsto [n']_\alpha, z \mapsto [n]_\alpha\}$ (with $n \neq n'$) has

$$[\![\texttt{<}x\texttt{>}z]\!]_V = [\texttt{<}n\texttt{>}n]_\alpha \neq [\texttt{<}n'\texttt{>}n]_\alpha = [\![\texttt{<}y\texttt{>}z]\!]_V .$$

Barendregt does not draw a distinction between names and schematic variables ranging over names, but the above example demonstrates that such a distinction does exist. If we have two distinct names n_1, n_2 then these will always be distinct, whereas two distinct schematic variables x_1, x_2 could be instantiated with the same name n by a valuation. We

have taken the more general route, permitting aliasing between variables of name sort. As we shall see, we can also model names which behave permutatively by imposing additional constraints that the variables must be mutually distinct (see Section 2.5.3 below).

This, along with point 3, means that it makes no sense to define the "free" variables of a pattern, because distinct variables might be identified by the valuation. The free names $FN(t)$ of the resulting α-tree is the only meaningful notion of "free names". $\diamond$

2.3.2 Schematic formulae and rules

In this section we use the meta-language of schematic patterns from the previous section to develop a language of schematic formulae and inductive rules. We first motivate our design choices by taking a high-level view of schematic inference rules.

Suppose that we wish to define n mutually recursive relations. We will fix *relation symbols* $r_1, \ldots, r_n$ with associated equality types $E_1, \ldots, E_n$. We write $r_i \subseteq E_i$ to mean that E_i is the equality type associated with r_i. A schematic rule for inductively defining such relations might take the form

$$\frac{r_j\, p_j \;\cdots\; r_k\, p_k \qquad c_1 \;\cdots\; c_m}{r_i\, p_i} \tag{2.1}$$

where $i, j, k \in \{1, \ldots, n\}$. The conclusion of (2.1) is an atomic formula where $r_i \subseteq E_i$ and where $\Delta \vdash p_i : E_i$ holds for some Δ. The premises consist of finite (possibly empty) lists of more atomic formulae ($r_j\, p_j, \ldots, r_k\, p_k$) and side-conditions ($c_1, \ldots, c_m$). The side-conditions are decidable constraints which restrict the set of instantiations that may be applied to a rule.

It is not immediately obvious what kinds of constraint c would give us a model of inductive definitions occurring in practice. It seems that the absolute minimum is constraints of *name inequality* $x \neq x'$, with x and x' being variables of the same name sort N. The need for these constraints arises from the fundamental asymmetry of pattern valuation—all occurrences of the same variable are always instantiated in the same way whereas different variables may also be instantiated the same, unless we explicitly state otherwise. For example, in a typical definition of substitution we might have as a base case:

$$\begin{array}{rcll} x[t/x] & \triangleq & t & \\ x'[t/x] & \triangleq & x' & \quad (\text{if } x \neq x'). \end{array}$$

In the second case, we must ensure that the schematic variables x and x' are always instantiated differently, to prevent the cases overlapping.

In practice it is worthwhile to generalise from name inequality constraints to *freshness constraints* $x \# p$ between a variable x of name sort and a pattern p of any equality type. This follows standard practice from nominal logic (Pitts, 2003) and nominal logic programming (Cheney and Urban, 2008). The intuition behind the freshness constraint is that the name x does not appear free in the term represented by p, in the sense of Definition 2.2.5. This will be formalised in Definition 2.4.4 below.

In the case when $\Delta \vdash p : N$, using Lemma 2.3.2 we know that p is actually a just variable x of name sort. Freshness constraints therefore subsume our original idea of name inequality constraints. They are sufficient for many purposes—in fact, Cheney and Urban have shown (Cheney and Urban, 2008) that inequality constraints between names are sufficient (indeed necessary) to inductively define full (α-)disequality for any nominal signature.

In addition to freshness constraints we add *equality constraints*. These boil down to α-equivalence constraints on the underlying set of ground trees. Adding equality constraints

$$\frac{\Delta \vdash p : E \qquad \Delta \vdash p' : E}{\Delta \vdash p = p'\ ok} \qquad \frac{\Delta \vdash p : N \qquad \Delta \vdash p' : E}{\Delta \vdash p \mathbin{\#} p'\ ok} \qquad \frac{\psi \in \{\mathsf{T}, \mathsf{F}\}}{\Delta \vdash \psi\ ok} \qquad \frac{r_i \subseteq E_i \qquad \Delta \vdash p : E_i}{\Delta \vdash r_i\, p\ ok}$$

$$\frac{\Delta \vdash \varphi\ ok \qquad \Delta \vdash \varphi'\ ok}{\Delta \vdash \varphi \mathbin{\&} \varphi'\ ok} \qquad \frac{\Delta \vdash \varphi\ ok \qquad \Delta \vdash \varphi'\ ok}{\Delta \vdash \varphi \vee \varphi'\ ok} \qquad \frac{x \notin dom(\Delta) \qquad \Delta, x : E \vdash \varphi\ ok}{\Delta \vdash \exists x : E.\, \varphi\ ok}$$

Figure 2.3: Typing rules for schematic constraints and formulae

does not provide any extra expressive power (since we could give an inductive definition that corresponds with the notion of α-equivalence for the nominal signature in question) but will make life much easier in Section 2.4 when we formally define the semantics of schematic inductive definitions.

Definition 2.3.6 (Schematic formulae). The set $Constr_\Sigma$ of atomic constraints is defined as follows.

$$\begin{array}{rcll} c \in Constr_\Sigma & ::= & p = p & \text{(equality constraint)} \\ & & p \mathbin{\#} p & \text{(freshness constraint).} \end{array}$$

These are used to build up the set of schematic formulae $Form_\Sigma$, which is defined by the following grammar.

$$\begin{array}{rcll} \varphi \in Form_\Sigma & ::= & r_i\, p & \text{(atomic formula)} \\ & & c & \text{(atomic constraint)} \\ & & \mathsf{T} & \text{(true)} \\ & & \mathsf{F} & \text{(false)} \\ & & \varphi \mathbin{\&} \varphi & \text{(conjunction)} \\ & & \varphi \vee \varphi & \text{(disjunction)} \\ & & \exists x : E.\, \varphi & \text{(existential).} \end{array}$$

Here, r_i is a member of our fixed, finite set of relation symbols $\{r_1, \ldots, r_n\}$. The only meta-level binder is in the existential form, where x is bound in the formula φ. ◇

Figure 2.3 presents rules defining a well-formedness judgement $\Delta \vdash \varphi\ ok$ for schematic formulae. The rules are standard—the case for an atomic formula assumes that the relation symbols $r_1, \ldots, r_n$ are associated with equality types $E_1, \ldots, E_n$ respectively. The side condition for the existential rule requires α-conversion at the meta-level in order to satisfy the side-condition that x be a fresh variable, and the rule for freshness constraints requires that the pattern on the left-hand side of the # be assigned a name sort N—by Lemma 2.3.2 this can only be satisfied if p is actually a variable x such that $\Delta(x) = N$.

We can now use schematic formulae to create inductive rules and full α-inductive definitions. Since the grammar of formulae includes atomic constraints (for side-conditions) and conjunctions, it is sufficient to consider schematic rules whose premise is a single formula φ.

Definition 2.3.7 (Schematic rules). A *schematic rule* has the form

$$\frac{\varphi}{r_i\, p} \tag{2.2}$$

where $i \in \{1, \ldots, n\}$. The rule in (2.2) is *well-formed* if there exists a type environment Δ such that $dom(\Delta) = vars(p)$ and such that $\Delta \vdash p : E_i$ and $\Delta \vdash \varphi\ ok$ both hold (it is not hard to see that

such an environment Δ exists and is unique). This means that any variables which appear in the premise but not in the conclusion must be existentially-quantified in the premise. ◇

Definition 2.3.8 (α-inductive definitions). An *α-inductive definition* $\mathcal{D}$ is a finite set of well-formed schematic rules. ◇

2.3.3 Example definition

In Figure 2.4 we continue our System F example. As far as possible, the rules use the same syntax as the grammar of System F terms given in Section 2.1. We use the variables x and a to stand for term variables and type variables respectively, and Γ to range over type environments, as is the convention. Figure 2.4 presents rules for an α-inductive definition of the following relations:

- **Looking up a variable in an environment:**

$$\mathtt{find} \subseteq \mathtt{tenv} * \mathtt{var} * \mathtt{type}$$

  where $\mathtt{find}\,(\Gamma\,,\,x\,,\,\tau)$ means that $\Gamma(x)$ is α-equivalent to τ. The definition of `find` consists of two rules—one for when the first variable in the environment matches the variable that we are looking for and one for when it does not (in which case we continue looking down the list). This illustrates the asymmetry of the traditional notion of pattern instantiation, as in the first rule we can express that the same name appears in two different places by using the variable x twice (a non-linear pattern) but we must use an explicit freshness constraint to ensure that the variables x and x' are always instantiated with distinct names. There is no rule for when the list is empty, as at this point we have failed to find the variable in question, so no derivation of the `find` formula exists. We could add the rule

$$\frac{\mathsf{F}}{\mathtt{find}\,(\mathtt{Nil}\,,\,x\,,\,\tau)}$$

  using the false formula, but this is redundant.

- **Capture-avoiding substitution of a type for a type variable throughout a type:**

$$\mathtt{ttsub} \subseteq \mathtt{type} * \mathtt{type} * \mathtt{tyvar} * \mathtt{type}$$

  where $\mathtt{ttsub}\,(\tau_1\,,\,\tau\,,\,\alpha\,,\,\tau_2)$ means that $\tau_1[\tau/\alpha]$ is α-equivalent to τ_2. The rules cover all of the cases of the nominal data sort `type`, with two rules for the base case (i.e. a type variable) as described above. The rule for universally-quantified types $\forall\alpha.\,\tau$ is interesting as it uses freshness constraints to require that the bound type variable a' must not be equal to the type variable a that is being substituted for, and furthermore that a' must not appear free in the type τ that is being substituted in. The substitution can only be pushed under the binder if these criteria are satisfied, which ensures that substitution is capture-avoiding.

- **System F typing judgement:**

$$\mathtt{type} \subseteq \mathtt{tenv} * \mathtt{term} * \mathtt{type}$$

  where $\mathtt{type}\,(\Gamma\,,\,M\,,\,\tau)$ means that $\Gamma \vdash M : \tau$ is provable using the rules of the System F type system. There is precisely one syntax-directed rule per constructor of the nominal data sort `term`. The case for a variable uses the `find` relation to look the variable up in the environment, and the cases for λ- and Λ-binders use freshness constraints to α-convert the bound

Rules for `find`**:**

$$\frac{\top}{\texttt{find}\,(\texttt{Cons}\,((x\,,\,\tau)\,,\,\Gamma)\,,\,x\,,\,\tau)} \qquad \frac{x\,\#\,x' \;\&\; \texttt{find}\,(\Gamma\,,\,x\,,\,\tau)}{\texttt{find}\,(\texttt{Cons}\,((x'\,,\,\tau')\,,\,\Gamma)\,,\,x\,,\,\tau)}$$

Rules for `ttsub`**:**

$$\frac{\top}{\texttt{ttsub}\,(\texttt{TyVar}\,a\,,\,\tau\,,\,a\,,\,\tau)} \qquad \frac{a\,\#\,a'}{\texttt{ttsub}\,(\texttt{TyVar}\,a'\,,\,\tau\,,\,a\,,\,\texttt{TyVar}\,a')}$$

$$\frac{\texttt{ttsub}\,(\tau_1\,,\,\tau\,,\,a\,,\,\tau_1') \;\&\; \texttt{ttsub}\,(\tau_2\,,\,\tau\,,\,a\,,\,\tau_2')}{\texttt{ttsub}\,(\texttt{Fun}\,(\tau_1\,,\,\tau_2)\,,\,\tau\,,\,a\,,\,\texttt{Fun}\,(\tau_1'\,,\,\tau_2'))} \qquad \frac{a'\,\#\,a \;\&\; a'\,\#\,\tau \;\&\; \texttt{ttsub}\,(\tau_1\,,\,\tau\,,\,a\,,\,\tau_2)}{\texttt{ttsub}\,(\texttt{ForAll}<a'>\tau_1\,,\,\tau\,,\,a\,,\,\texttt{ForAll}<a'>\tau_2)}$$

Rules for `type`**:**

$$\frac{\texttt{find}\,(\Gamma\,,\,x\,,\,\tau)}{\texttt{type}\,(\Gamma\,,\,\texttt{Var}\,x\,,\,\tau)} \qquad \frac{\texttt{type}\,(\texttt{Cons}\,((x\,,\,\tau_1)\,,\,\Gamma)\,,\,M\,,\,\tau_2) \;\&\; x\,\#\,\Gamma}{\texttt{type}\,(\Gamma\,,\,\texttt{Lam}<x>(\tau_1\,,\,M)\,,\,\texttt{Fun}\,(\tau_1\,,\,\tau_2))}$$

$$\frac{\texttt{type}\,(\Gamma\,,\,M_1\,,\,\texttt{Fun}\,(\tau_1\,,\,\tau_2)) \;\&\; \texttt{type}\,(\Gamma\,,\,M_2\,,\,\tau_1)}{\texttt{type}\,(\Gamma\,,\,\texttt{App}\,(M_1\,,\,M_2)\,,\,\tau_2)} \qquad \frac{\texttt{type}\,(\Gamma\,,\,M\,,\,\tau) \;\&\; a\,\#\,\Gamma}{\texttt{type}\,(\Gamma\,,\,\texttt{Gen}<a>M\,,\,\texttt{ForAll}<a>\tau)}$$

$$\frac{\texttt{type}\,(\Gamma\,,\,M\,,\,\texttt{ForAll}<a>\tau_1) \;\&\; \texttt{ttsub}\,(\tau_1\,,\,\tau_2\,,\,a\,,\,\tau)}{\texttt{type}\,(\Gamma\,,\,\texttt{Spec}\,(M\,,\,\tau_2)\,,\,\tau)}$$

Figure 2.4: Example α-inductive definitions for the System F type system

variables (x and a respectively) so that they do not appear in Γ. The rule for the type specialisation operation introduces a new name a of sort `tyvar` on the top line, which appears both in abstraction position and also free. The issue of whether this freshly-generated type variable escapes its scope (i.e. appears free in the conclusion of the rule instance) is related to work carried out on "variable-convention compatible" inductive definitions (Urban et al., 2007).

It is worth pointing out that α-inductive definitions are not just restricted to encoding operational semantics and type systems. The syntax is very general and can encode a great many systems: the only proviso is that they can be expressed in an inference rule format.

2.4 Semantics of α-inductive definitions

In this section we present a straightforward term-model semantics for α-inductive definitions in terms of α-equivalence classes of ground trees.

2.4.1 Simplifying definitions

We have already seen one simplification in the presentation of α-inductive definitions—since the grammar of formulae contains conjunction it suffices to only consider rules with a single formula φ on the top line. However, Definition 2.3.6 introduces more kinds of formulae which do not contribute any extra expressive power—in particular, equality constraints, disjunction

and existential quantification. The motivation for including these is to further restrict the class of α-inductive definitions that we must consider. This will greatly simplify the presentation and the proofs.

Moving to a single relation symbol

At the expense of extending the nominal signature we can consider α-inductive definitions where the rules each contain a single relation symbol—we will fix the relation symbol r for this purpose.

Suppose we have a nominal signature Σ and an α-inductive definition $\mathcal{D}$ concerning relation symbols $r_1, \ldots, r_n$, which represent subsets of the equality types $E_1, \ldots, E_n$ respectively. We extend Σ to produce a new signature Σ', which is related to the original signature as follows.

- $\mathbb{N}_{\Sigma'} \triangleq \mathbb{N}_{\Sigma}$
- $\mathbb{S}_{\Sigma'} \triangleq \mathbb{S}_{\Sigma} \uplus \{S_r\}$
- $\mathbb{C}_{\Sigma'} \triangleq \mathbb{C}_{\Sigma} \uplus \{\mathbb{R}_1 : E_1 \to S_r, \ldots, \mathbb{R}_n : E_n \to S_r\}$.

We add a new nominal data sort to represent inductively-defined relations, which we will refer to as S_r. This new sort must not appear either in $\mathbb{S}_{\Sigma}$ or $\mathbb{N}_{\Sigma}$. We then represent the original relation symbols $r_1, \ldots, r_n$ using n new, distinct constructors $\mathbb{R}_1, \ldots, \mathbb{R}_n$, whose argument types correspond to the types of the original relations. Therefore the relation symbol r represents an α-tree relation in the following sense.

Definition 2.4.1 (α-tree relations). An *α-tree relation* is a subset $R \subseteq \alpha\text{-}\mathit{Tree}_{\Sigma'}(S_r)$ ◇

The schematic rules are altered by replacing every atomic formula $r_i\, p$ by the atomic formula $r\,(\mathbb{R}_i\, p)$. The task of matching against the relation symbol in an atomic formula is now done by normal pattern-matching on schematic patterns.

This transformation exploits the fact that subsets $R \subseteq \alpha\text{-}\mathit{Tree}_{\Sigma}(S_r)$ are in bijection with n-tuples of subsets $R_1 \subseteq \alpha\text{-}\mathit{Tree}_{\Sigma}(E_1), \ldots, R_n \subseteq \alpha\text{-}\mathit{Tree}_{\Sigma}(E_n)$. Hence, the semantics of the original α-inductive definition is preserved (though we do not define it formally).

Moving to a single rule

We can go even further—because of the presence of equality, disjunction and existential quantification in our grammar of formulae it actually suffices to consider α-inductive definitions that consist of a single rule involving a single relation symbol.

To illustrate how this translation occurs, suppose that we have a definition $\mathcal{D}$ which uses a single relation symbol r but has n rules:

$$\frac{\varphi_1}{r\,p_1} \qquad \frac{\varphi_2}{r\,p_2} \qquad \cdots \qquad \frac{\varphi_{n-1}}{r\,p_{n-1}} \qquad \frac{\varphi_n}{r\,p_n}$$

The presence of multiple rules is an implicit disjunction, as we can make progress using any rule if none of the others are applicable. This suggests that we may able to combine the premises of the rules using an n-way disjunction, but to do so we must provide a single conclusion for the combined rule.

The answer comes from equality constraints and existential quantification. We fix a new variable x which does not occur in the rules for $\mathcal{D}$ presented above, and use the atomic formula $r\,x$ as the conclusion of the combined rule. The variable x stands for the ground predicate

instance for which we are trying to construct a derivation. To construct the corresponding premise (for the ith rule above) we existentially quantify the variables which occur in p_i and require that x is α-equivalent to p_i. We can then process the original premise φ_i as normal. The rules for $\mathcal{D}$ would therefore become the following single rule

$$\frac{(\exists vars(p_1).\, x = p_1 \ \&\ \varphi_1) \vee \cdots \vee (\exists vars(p_n).\, x = p_n \ \&\ \varphi_n)}{r\, x} \tag{2.3}$$

where we write $\exists vars(p).\, \varphi$ for the iterated $\exists$-quantification of all of the variables appearing in p (with the corresponding type annotations for the variables in $vars(p)$).

The rule (2.3) constitutes a new α-inductive definition $\mathcal{D}'$ whose semantics is identical to that of $\mathcal{D}$. This is a straightforward consequence of the semantics of schematic formulae (defined in the next section) so we omit the proof.

Definition 2.4.2 (Standard α-inductive definitions). An α-inductive definition $\mathcal{D}$ in *standard form* of a set of α-trees of equality type S_r is given by the single inference rule

$$\frac{\varphi}{r\, x} \tag{2.4}$$

where $\{x : S_r\} \vdash \varphi\ ok$ holds (i.e. x is the only meta-level free variable in φ), and r is the only relation symbol that appears in φ. ◇

Henceforth, we will only consider α-inductive definitions in standard form. We illustrate the process of transforming α-inductive definitions into standard form with a small definition from our running System F example.

Example 2.4.3 (Free type variables in a System F type). The four rules below define a relation $\mathtt{ftv} \subseteq \mathtt{tyvar} * \mathtt{type}$ which encodes the freshness relation between a type variable and a System F type.

$$\frac{a \,\#\, a'}{\mathtt{ftv}\,(a\,,\,\mathtt{TyVar}\,a')} \qquad \frac{\mathtt{ftv}\,(a\,,\,\tau_1)\ \&\ \mathtt{ftv}\,(a\,,\,\tau_2)}{\mathtt{ftv}\,(a\,,\,\mathtt{Fun}\,(\tau_1\,,\,\tau_2))}$$

$$\frac{\top}{\mathtt{ftv}\,(a\,,\,\mathtt{ForAll}\,{<}a{>}\tau)} \qquad \frac{a \,\#\, a'\ \&\ \mathtt{ftv}\,(a\,,\,\tau)}{\mathtt{ftv}\,(a\,,\,\mathtt{ForAll}\,{<}a'{>}\tau)}$$

The intuition is that $\mathtt{ftv}\,(a\,,\,\tau)$ means that the type variable a does not appear free (i.e. not under a $\forall$-binder) in the System F type τ. This definition is redundant as the same effect could be achieved by using the built-in freshness constraint $a \,\#\, \tau$, but it is a good example because it is small and self-contained. We will write $\mathcal{D}_{\text{ftv}}$ for the α-inductive definition over $\mathcal{F}$ consisting of just the four rules above.

In order to convert $\mathcal{D}_{\text{ftv}}$ into standard form we must first extend the nominal signature $\mathcal{F}$ from Figure 2.1. We add a new nominal data sort S_r and a new constructor `ftv`, with the following type.

$$\mathtt{ftv} : \mathtt{tyvar} * \mathtt{type} \to S_r$$

We can now perform the transformations outlined above to produce a new inductive definition

$\mathcal{D}'_{\text{ftv}}$ over $\mathcal{F}'$ which is in standard form.

$$\frac{\begin{array}{ll} & (\exists a\texttt{:tyvar}.\, \exists a'\texttt{:tyvar}.\, x = \texttt{ftv}\;(a\texttt{,}\;\texttt{TyVar}\;a')\;\&\;a \,\#\, a') \\ \vee & (\exists a\texttt{:tyvar}.\, \exists \tau_1\texttt{:type}.\, \exists \tau_2\texttt{:type}. \\ & \quad x = \texttt{ftv}\;(a\texttt{,}\;\texttt{Fun}\;(\tau_1\texttt{,}\;\tau_2))\;\&\;r\;(\texttt{ftv}\;(a\texttt{,}\;\tau_1))\;\&\;r\;(\texttt{ftv}\;(a\texttt{,}\;\tau_2))) \\ \vee & (\exists a\texttt{:tyvar}.\, \exists \tau\texttt{:type}.\, x = \texttt{ftv}\;(a\texttt{,}\;\texttt{ForAll}\;\texttt{<}a\texttt{>}\tau)\;\&\;\top) \\ \vee & (\exists a\texttt{:tyvar}.\, \exists a'\texttt{:tyvar}.\, \exists \tau\texttt{:type}.\, x = \texttt{ftv}\;(a\texttt{,}\;\texttt{ForAll}\;\texttt{<}a'\texttt{>}\tau)\;\&\;a \,\#\, a'\;\&\;r\;(\texttt{ftv}\;(a\texttt{,}\;\tau))) \end{array}}{r\;x}$$

Even with this small example, it is apparent that the blowup in the size of the schematic rule makes α-inductive definitions in standard form too unwieldy for humans to deal with. ◇

2.4.2 Semantics of formulae

The first step towards providing a semantics for α-inductive definitions is to define satisfaction of atomic constraints. It is straightforward to show that

$$\Delta \vdash p : E \Longrightarrow r \notin \mathit{vars}(p) \tag{2.5}$$

where $r \subseteq S_r$, which implies that the relation symbol r may not appear in atomic constraints. Hence, the satisfaction relation has the form $V \models c$ as we need a valuation V to instatiate any variables occurring in c but do not need to consider the semantics of the relation r.

Definition 2.4.4 (Satisfaction of atomic constraints). If $\Delta \vdash c\ \mathit{ok}$ and $V \in \alpha\text{-}\mathit{Tree}_\Sigma(\Delta)$ then we define satisfaction of atomic constraints by cases, as follows.

$$\begin{array}{rcl} V \models p = p' & \Longleftrightarrow & [\![p]\!]_V = [\![p']\!]_V \\ V \models x \,\#\, p & \Longleftrightarrow & V(x) \not\approx [\![p]\!]_V . \end{array}$$ ◇

This simplicity of this definition stems from the fact that patterns denote α-equivalence classes. This means that α-equivalence is handled implicitly and we do not need to resort to technical devices such as permutations, as in other work on nominal abstract syntax (Urban et al., 2004; Cheney, 2004b; Shinwell, 2005).

We now consider the semantics of schematic formulae. As for atomic constraints, the satisfaction judgement must involve a valuation V with the appropriate domain. However, because the relation symbol r may appear in formulae, we need to interpret it using an α-tree relation R.

Definition 2.4.5 (Satisfaction of formulae). If $r \subseteq S_r$, $\Delta \vdash \varphi\texttt{:prop}$, $V \in \alpha\text{-}\mathit{Tree}_\Sigma(\Delta)$ and $R \subseteq \alpha\text{-}\mathit{Tree}_\Sigma(S_r)$, then satisfaction of formulae is written $(R, V) \models \varphi$ and is defined by the rules in Figure 2.5. We write $V[x \mapsto t]$ for the valuation which has domain $\mathit{dom}(V) \uplus \{x\}$, maps x to t and otherwise behaves like V. ◇

Again, α-equivalence is handled implicitly but formally. Most of these rules are completely standard, which is one of the advantages of our approach. The rule for existential formulae $\exists x : E.\, \varphi$ has the hypothesis that there must exist a ground tree $t \in \alpha\text{-}\mathit{Tree}_\Sigma(E)$. Therefore, the judgement $(R, V) \models \exists x : E.\, \varphi$ cannot hold if there do not exist any ground trees of type E. For example, it is possible to make circular datatype definitions such as the following.

```
datatype foo = K of foo
```

$$\frac{[\![p]\!]_V \in R}{(R,V) \models r\,p} \qquad \frac{V \models c}{(R,V) \models c} \qquad \frac{(R,V) \models \varphi_1 \qquad (R,V) \models \varphi_2}{(R,V) \models \varphi_1 \mathbin{\&} \varphi_2}$$

$$\frac{}{(R,V) \models \top} \qquad \frac{(R,V) \models \varphi_1}{(R,V) \models \varphi_1 \vee \varphi_2} \qquad \frac{(R,V) \models \varphi_2}{(R,V) \models \varphi_1 \vee \varphi_2}$$

$$\frac{x \notin dom(V) \qquad t \in \alpha\text{-}Tree_\Sigma(E) \qquad (R, V[x \mapsto t]) \models \varphi}{(R,V) \models \exists x\!:\!E.\,\varphi}$$

Figure 2.5: Formula satisfaction rules

Clearly there cannot exist any (finite) ground trees of type `foo`, so any existential formula over this type cannot be satisfiable. We will return to this point in Section 3.3.2.

The rule for atomic formulae requires that the instantation $[\![p]\!]_V$ of the pattern p is a member of the α-tree relation R, which again relies on the fact that the relation symbol r cannot appear as a variable in the pattern p—this means that we can produce a ground α-tree from p by performing the instantiation $[\![p]\!]_V$. By using variables that range over α-equivalence classes directly we build in α-equivalence from the ground up.

Lemma 2.4.6 (Monotonicity). *If $R \subseteq R' \subseteq \alpha\text{-}Tree_\Sigma(S_r)$ and $(R,V) \models \varphi$ then $(R',V) \models \varphi$.* □

2.4.3 Semantics of definitions

Recall that we only consider α-inductive definitions $\mathcal{D}$ which are in standard form in the sense of Definition 2.4.2.

Definition 2.4.7. The *denotation* $[\![\mathcal{D}]\!] \subseteq \alpha\text{-}Tree_\Sigma(S_r)$ of a standard α-inductive definition $\mathcal{D}$ (as in Definition 2.4) is the least fixed point of the monotone function $\Phi_{\mathcal{D}}$ on subsets of α-trees, defined by

$$\Phi_{\mathcal{D}}(R) \triangleq \{t \in \alpha\text{-}Tree_\Sigma(S_r) \mid (R, \{x \mapsto t\}) \models \varphi\}. \tag{2.6}$$

The notation $\{x \mapsto t\}$ represents the valuation V which has $dom(V) = \{x\}$ and maps that variable to the α-tree t. The least fixed point exists by Tarski's fixed point theorem (Tarski, 1955). To show that $\Phi_{\mathcal{D}}$ is monotone we use Lemma 2.4.6, which relies on the fact that the relation symbol r only appears positively in φ. ◇

The definition of $[\![\mathcal{D}]\!]$ by way of (2.6) is a precise way of stating the informal view on the semantics of inductively-defined relations:

- *rules are schematic*—(2.4) has the variable x in its conclusion;
- *we instantiate the rules to produce ground instances*—this is the effect of the $\{x \mapsto t\}$ valuation in (2.6); and
- *we take the "least set closed under the rules"*—this is the least R such that $\Phi_{\mathcal{D}}(R) \subseteq R$, i.e. the least fixed point of $\Phi_{\mathcal{D}}$.

Furthermore, $\Phi_{\mathcal{D}}$ is finitary which means that we can construct $[\![\mathcal{D}]\!]$ as the union of a chain of subsets of $\alpha\text{-}Tree_\Sigma(S_r)$, as illustrated by the following Lemma.

Lemma 2.4.8. *For any α-inductive definition $\mathcal{D}$ in standard form, we can construct $[\![\mathcal{D}]\!]$ as*

$$[\![\mathcal{D}]\!] = \bigcup_{n \in \mathbb{N}} [\![\mathcal{D}]\!]^{(n)} \tag{2.7}$$

where $[\![\mathcal{D}]\!]^{(n)}$ is the n-fold application $\Phi^n_{\mathcal{D}}(\emptyset)$.

Proof. Since the rules in Definition 2.4.5 each only have finitely many hypotheses, $\Phi_{\mathcal{D}}$ is a finitary monotone operation and hence we get (2.7). □

The brevity of this presentation is one of its main advantages. The framework is simple, easy to explain and yet is fairly expressive and handles binding in a reasonably intuitive way.

2.5 α-inductive definitions and equivariance

In the final section of this chapter we study the relationship of α-inductive definitions to the concept of equivariance from nominal logic.

2.5.1 Permutations

Up to now, we have hardly mentioned name-permutations, which are a staple of most nominal techniques for abstract syntax involving binders (Gabbay and Pitts, 2002; Pitts, 2003; Cheney and Urban, 2008). They are not required for our approach because α-equivalence is handled by the explicit use of α-equivalence classes of ground trees. Furthermore, our approach to representing binders as meta-variables as opposed to permutative names is more general than existing approaches, as different bound variables may be "aliased" to stand for the same underlying name.

As our approach is more general it should be able to simulate the permutative names. In the rest of this chapter we will introduce permutations of names and use them to encode existing nominal approaches in our system.

Definition 2.5.1 (Permutations). Permutations $\pi \in \mathit{Perm}$ are bijections from *Name* to *Name* which are *finite* (the set $\{n \in \mathit{Name} \mid \pi(n) \neq n\}$ is finite) and *sort-respecting* ($\mathit{sort}(\pi(n)) = \mathit{sort}(n)$ for all $n \in \mathit{Name}$). ◇

Definition 2.5.2. Given a ground tree $g \in \mathit{Tree}_\Sigma(E)$, the notation $\pi \cdot g \in \mathit{Tree}_\Sigma(E)$ denotes the result of permuting all names occurring in g according to π. Since this action respects α-equivalence, we get a well-defined action on α-trees that satisfies

$$\pi \cdot [g]_\alpha = [\pi \cdot g]_\alpha. \tag{2.8}$$

◇

Equipped with this permutation action, it is not hard to see that the set of ground trees Tree_Σ is a nominal set (Pitts, 2003).

2.5.2 Equivariance

Equivariance is a fundamental assumption underlying nominal logic. It is described in (Pitts, 2003), which states that

> *the only predicates we ever deal with (when describing properties of syntax) are equivariant ones, in the sense that their validity is invariant under swapping (i.e., transposing, or interchanging) names.*

Swapping is used in the definition because it has far better logical properties than capture-avoiding renaming. This is because it is a self-inverse function, i.e. $(a\,a') \cdot ((a\,a') \cdot t) = t$, which means that the class of equivariant predicates is closed under (amongst other things) negation, conjunction, $\forall$- and $\exists$-quantification, formation of least and greatest fixed points of monotone operators (Pitts, 2003). The relationship between name-swappings and name-permutations (as defined in Definition 2.5.1) is via group theory—it can be shown that any finite permutation can be represented as a composition of individual swappings.

Why is equivariance fundamental? Why was Barendregt justified to introduce variables v_0, $v_1, \ldots$ which are thereafter ignored and only referred to via the meta-variables $x, y, z, \ldots$? The reason is that *it should be irrelevant which actual names are used*. As we highlighted in Remark 2.3.5 there are subtle issues to bear in mind when representing abstract syntax involving binders in this manner.

We take the view that concrete names are an implementation detail that should be hidden from the programmer if at all possible. Similarly, a compiler writer might care at what particular memory address a piece of data is stored, whereas a programmer using the compiler should not need to worry about such issues.

Definition 2.5.3 (Equivariant α-tree relations). An α-tree relation $R \subseteq \alpha\text{-}Tree_\Sigma(S_r)$ is *equivariant* if $R \subseteq \pi \cdot R$ for all π, where

$$\pi \cdot R \triangleq \{\pi \cdot t \mid t \in R\}. \tag{2.9}$$

◇

Informally, an equivariant α-tree relation is one that is closed under permutation of names. This means that the membership of α-equivalence class t in the relation is not dependent on the particular names that occur within the representatives of the α-equivalence class. We now make some definitions and prove some intermediate results in preparation for the main result of this section, which is that all inductive definitions denote equivariant relations on α-trees.

Definition 2.5.4. Given an α-tree valuation $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ and a permutation π, we write $\pi \cdot V$ for the α-tree valuation in $\alpha\text{-}Tree_\Sigma(\Delta)$ which maps x to $\pi \cdot t$ if $V(x) = t$. ◇

Lemma 2.5.5. *For any $\Delta \vdash p : E$, $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ and $\pi \in Perm$ we have $[\![p]\!]_{(\pi \cdot V)} = \pi \cdot ([\![p]\!]_V)$.*

Proof. By induction on the structure of p, using the defining properties of the mapping $p \mapsto [\![p]\!]_V$ from Lemma 2.3.4. □

Lemma 2.5.6 (Equivariance of satisfaction). *For any constraint $\Delta \vdash c$ ok, formula $\Delta \vdash \varphi$ ok, α-tree relation $R \subseteq \alpha\text{-}Tree_\Sigma(S_r)$, valuation $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ and permutation $\pi \in Perm$, we have*

$$V \models c \implies \pi \cdot V \models c \tag{2.10}$$

$$(R, V) \models \varphi \implies (\pi \cdot R, \pi \cdot V) \models \varphi. \tag{2.11}$$

Proof. Property (2.10) follows from Lemma 2.5.5 and the fact that $\pi \cdot (FN(t)) = FN(\pi \cdot t)$. Property (2.11) follows by induction on the structure of φ, using the definition of $(R, V) \models \varphi$ from Definition 2.4.5; Lemma 2.5.5 is needed in the case where φ is $r\,p$ and property (2.10) is needed in the case where φ is an atomic constraint c. □

We now prove the main result in this section.

Theorem 2.5.7. *The denotation $[\![\mathcal{D}]\!] \subseteq \alpha\text{-}Tree_\Sigma(S_r)$ of any α-inductive definition $\mathcal{D}$ is an equivariant α-tree relation.*

Proof. Suppose that $\mathcal{D}$ is defined by a single schematic inference rule as in (2.4). By definition $[\![\mathcal{D}]\!]$ is the least subset $R \subseteq \alpha\text{-}Tree_\Sigma(S_r)$ closed under $\Phi_\mathcal{D}$, that is, satisfying

$$\{t \in \alpha\text{-}Tree_\Sigma(S_r) \mid (R, \{x \mapsto t\}) \models \varphi\} \subseteq R. \tag{2.12}$$

But if any R satisfies (2.12), so does $\pi^{-1} \cdot R$, for if $(\pi^{-1} \cdot R, \{x \mapsto t\}) \models \varphi$ then by (2.11) we have $(\pi\pi^{-1} \cdot R, \{x \mapsto \pi \cdot t\}) \models \varphi$, i.e. $(R, \{x \mapsto \pi \cdot t\}) \models \varphi$; so by (2.12) we have $\pi \cdot t \in R$ and hence $t \in \pi^{-1} \cdot R$. So in particular $\pi^{-1} \cdot [\![\mathcal{D}]\!]$ is a relation closed under $\Phi_\mathcal{D}$ and hence $[\![\mathcal{D}]\!] \subseteq \pi^{-1} \cdot [\![\mathcal{D}]\!]$. Applying π to both sides gives us $\pi \cdot [\![\mathcal{D}]\!] \subseteq [\![\mathcal{D}]\!]$, as required. □

This result implies that users cannot write down α-inductive definitions whose meaning depends on particular names. We would expect this result to hold since there are no names in the syntax of α-inductive definitions (see Remark 2.3.5, comment 4), so a schematic variable x of name sort could be instantiated with *any* name α-equivalence class $[n]_\alpha$ (of the correct sort).

Theorem 2.5.7 formalises that the abstraction boundary between the user's view (of names as meta-variables) and the internal view (where the denotation of an α-inductive definitions is a set of α-equivalence classes over a term language involving particular ground names) can never be breached.

2.5.3 Some/any property of satisfaction

Although distinct variables of name sort may actually be instantiated with the same concrete name, if we impose sufficient distinctions between the variables then we can deduce a "some/any" property of formula satisfaction reminiscent of those commonplace in work on nominal logic, for example in (Gabbay and Pitts, 2002, Proposition 4.10), (Pitts, 2003, Proposition 4) and (Pitts, 2006, Theorem 3.8). We first define a shorthand for the constraints required to assert that a set of variables (of name sort) be mutually distinct.

Definition 2.5.8 (Name distinction constraints). Fix a set $\overline{x}$ of (distinct) variables $x_1, \ldots, x_n$. Then, define $\#_{\overline{x}}$ to be the set of atomic constraints

$$\#_{\overline{x}} \triangleq \{x_i \# x_j \mid 1 \leq i < j \leq n\}. \tag{2.13}$$

◇

Lemma 2.5.9 (Some properties of distinctness constraints).

- $\Delta \vdash \#_{\overline{x}}$ *ok holds if, for all* $x \in \overline{x}$, $x \in dom(\Delta)$ *and* $\Delta(x) = N$ *for some name sort* N.
- *If the list* $\overline{x}$ *contains no duplicate variables then the constraint* $\#_{\overline{x}}$ *is satisfiable (although of course it may not be satisfiable in the presence of other constraints).*
- *If* $V \models \#_{\overline{x},\overline{x}'}$ *then* $V \models \#_{\overline{x}}$. □

The following Theorem expresses the "some/any" property for schematic formulae. If φ is a formula which only contains variables $\overline{x}$ of name sorts $\overline{N}$, then it is satisfied by some valuation that satisfies $\#_{\overline{x}}$ if and only if it is satisfied by any such valuation.

Theorem 2.5.10. *Suppose that* $\Delta \vdash \varphi$ *ok, where* Δ *maps distinct variables* $\overline{x}$ *to name sorts* $\overline{N}$. *Then the following are equivalent:*

$$\forall V \in \alpha\text{-}Tree_\Sigma(\Delta).\ V \models \#_{\overline{x}} \implies ([\![\mathcal{D}]\!], V) \models \varphi \tag{2.14}$$

$$\exists V \in \alpha\text{-}Tree_\Sigma(\Delta).\ V \models \#_{\overline{x}} \wedge ([\![\mathcal{D}]\!], V) \models \varphi. \tag{2.15}$$

Proof. We prove the two directions of the equivalence separately.

(2.14) $\Longrightarrow$ (2.15). Since the set $\{V \mid V \in \alpha\text{-}\mathit{Tree}_\Sigma(\Delta) \wedge V \models \#_{\overline{x}}\}$ is non-empty (in fact infinite) it follows that (2.14) implies (2.15).

(2.15) $\Longrightarrow$ (2.14). Suppose there exists a valuation $V \in \alpha\text{-}\mathit{Tree}_\Sigma(\Delta)$ such that $V \models \#_{\overline{x}}$ and $(\llbracket \mathcal{D} \rrbracket, V) \models \varphi$ both hold. Then, given any other $V' \in \alpha\text{-}\mathit{Tree}_\Sigma(\Delta)$ such that $V' \models \#_{\overline{x}}$, we must show that $(\llbracket \mathcal{D} \rrbracket, V') \models \varphi$. But, if $V \models \#_{\overline{x}}$ and $V' \models \#_{\overline{x}}$ both hold, it must be the case that V' is $\pi \cdot V$ for some $\pi \in \mathit{Perm}$. Then, from $(\llbracket \mathcal{D} \rrbracket, V) \models \varphi$ and Lemma 2.5.6 we get that $(\pi \cdot \llbracket \mathcal{D} \rrbracket, \pi \cdot V) \models \varphi$. Finally, by Theorem 2.5.7 and the fact that V' is $\pi \cdot V$ we get $(\llbracket \mathcal{D} \rrbracket, V') \models \varphi$, as required. □

Chapter 3

αML

"Haben Sie einen gesehen?"
—E. Mach

In this chapter we define the meta-language αML, which will serve as a host language for encoding inductive definitions of the form presented in Chapter 2. We will first present the syntax and static semantics of αML, before discussing the satisfaction of constraints. We will then present a small-step operational semantics of the language, phrased in terms of frame stacks as in (Pitts, 2005), and demonstrate a big-step termination relation defined in terms of the small-step one. We will show that αML is conservative over a traditional higher-order typed functional programming language—the proof that a constraint logic programming language (over the domain of α-trees) is also embedded in the language is deferred to Chapter 4. The operational semantics will be used in Chapter 5 when we prove properties of observational equivalence of αML expressions.

3.1 Language overview

αML is a call-by-value, higher-order, typed meta-programming language. It includes the basic features required to implement α-inductive definitions, namely:

- the ability to create and deconstruct name abstractions;
- support for generating names and for name-aliasing;
- constraints of equality and name-freshness; and
- fair proof-search by non-deterministic branching.

αML unites features typically found in both functional and constraint logic programming (Jaffar et al., 1998) languages. From the functional side we inherit recursive functions and the ability to define functions by recursion over the structure of some datatype. The features of constraint logic programming found in αML include the ability to assert α-equivalence and name-freshness constraints on object-level terms and a high-level branching operator which implements fair proof-search as a language primitive.

Perhaps the most important feature of αML is its representation of object-level names and binding. As in the language of schematic patterns and formulae from Chapter 2, object-level names are represented using variables which may be aliased. This is in direct contrast to existing systems such as FreshML (Shinwell, 2005; Shinwell et al., 2003) and αProlog (Cheney and Urban, 2004) where permutative names in the meta-language are used to represent object-level

names. As permutative names have their own unique identity, examples of aliasing such as those from Section 1.4.2 do not arise in those languages.

FreshML and αProlog both use the И-quantifier from nominal logic (Pitts, 2003) to generate a fresh name that has not been seen before (in FreshML this is written using the `fresh` keyword). In αML, however, variables of name sort are generated using the same ∃-quantifier which generates variables to stand for data terms. This same quantifier is only used in αProlog to generate data variables, whereas FreshML does not include any logic programming facilities and therefore does not include an ∃-quantifier. Finally, αML follows other nominal metaprogramming languages by using the $<v>v'$ term-former to represent binding of names in the object-language.

Historical note 3.1.1 (Names in MLSOS). The MLSOS language described in (Lakin and Pitts, 2008) was a precursor to αML. MLSOS contained three different kinds of name:

1. *value identifiers* ranging over meta-language values;
2. *permutative atoms* standing for object-level names; and
3. *logic variables* standing for unknown object-level terms.

In MLSOS, logic variables could be created at name sorts (to stand for unknown names) and constraints of equality and disequality could be expressed between these. However, these logic variables were not permitted to appear in binding position, by the syntax of the language. In αML we do away with the final two categories of name listed above, so meta-level variables stand for meta-level values as well as unknown object-level terms. This is akin to executing code with unbound variables. Unlike in MLSOS, variables may appear in binding position. This gives us more flexibility and permits aliasing between bound names, which MLSOS and other nominal programming languages cannot encode. It also more closely models informal practice in the definition of the syntax and semantics of programming languages and calculi. However, this design choice complicates the associated constraint problem, as we shall see in Section 3.4 and Chapter 6. ◇

3.2 αML syntax

We will use the same countably infinite set *Var* of variables, ranged over by x, that was introduced in Section 2.3.1. This will greatly simplify the translation between the syntax of schematic formulae and that of αML.

For a fixed nominal signature Σ, we will write Val_Σ, $Constr_\Sigma$ and Exp_Σ for the sets of αML values, constraints and expressions respectively. These are generated by the grammar in Figure 3.1. We note that $Val_\Sigma \subset Exp_\Sigma$ and that $Constr_\Sigma$ is contained in the grammar of formulae defined in Chapter 2. The meta-variable T ranges over αML types, which will be introduced in Section 3.3.

The grammar in Figure 3.1 restricts the class of valid αML expressions to a form similar to the A-normal form of (Flanagan et al., 1993), where evaluation order is specified using `let` bindings. This simplifies the presentation and makes proofs more straightforward, without reducing the expressiveness of the language—for example, a more general language construct such as the application $e\,e'$ can be encoded in A-normal form as

$$\texttt{let}\ x = e\ \texttt{in}\ \texttt{let}\ x' = e'\ \texttt{in}\ x\,x'$$

where x and x' are two distinct, freshly-chosen variables. In the translation to A-normal form the addition of extra `let` bindings makes the evaluation order completely explicit.

$v \in Val_\Sigma$	$::=$	x, f	(variable)
		$K\,v$	(constructor application)
		`()`	(unit)
		$(v, \ldots, v)$	(tuple)
		$\texttt{fun}\, f\, (x : T) : T = e$	(recursive function)
		$\top$	(success)
		$\langle v \rangle v$	(name abstraction)
$c \in Constr_\Sigma$	$::=$	$v = v$	(equality constraint)
		$v \,\#\, v$	(freshness constraint)
$e \in Exp_\Sigma$	$::=$	v	(value)
		$\texttt{let}\; x = e \;\texttt{in}\; e$	(`let` binding)
		$v\,v$	(function application)
		$\texttt{case}\; v \;\texttt{of}\; K\,x \texttt{->} e \mid \cdots \mid K\,x \texttt{->} e$	(`case` expression)
		$v.i$	(projection)
		c	(constraint)
		$\exists x : E.\, e$	(existential)
		$e \parallel e$	(non-deterministic branch).

Figure 3.1: αML values, expressions and constraints

We identify expressions up to α-conversion of bound variables. The binding forms in the meta-language are as follows, where the bound variable(s) and their corresponding scope(s) are underlined.

$$\texttt{fun}\, \underline{f}\, (\underline{x} : T) : T' = \underline{e} \qquad \texttt{let}\; \underline{x} = e \;\texttt{in}\; \underline{e'} \qquad \texttt{case}\; v \;\texttt{of}\; K_1\, \underline{x_1} \texttt{->} \underline{e_1} \mid \cdots \mid K_n\, \underline{x_n} \texttt{->} \underline{e_n} \qquad \exists \underline{x} : E.\, \underline{e}$$

For `case` expressions, the variable x_1 is bound in e_1 only, and so on. We write $FV(e)$ for the set of free variables of the expression e.

The syntax in Figure 3.1 corresponds to a traditional functional programming language with `case` expressions, tuple projection and recursive functions, extended with the novel constructs described in Section 1.4. We now discuss those novel constructs, beginning with the $\top$ value. This is a dummy value which represents successful completion of some logic programming computation, potentially involving proof-search and non-determinism.

The abstraction term-former $\langle v \rangle v'$ represents an object-level name-binder, with the single name v bound in its lexical scope v'. This is important because α-equivalence in the object-language arises between binders modelled using this meta-level syntax. Once again, we reiterate that the abstraction term-former is not a binder in the meta-language. This means that we regard $\langle x \rangle x$ and $\langle y \rangle y$ as distinct expressions when $x \neq y$. However, in Chapter 5 we show that any two expressions which represent α-equivalent terms in the object-language are observationally equivalent. The type system presented in Section 3.3.3 will ensure that the value in abstraction position is always a variable x of name sort.

The constraints in the αML syntax are identical to those introduced in Definition 2.3.6, namely equality (which corresponds to object-level α-equivalence) and freshness (which corresponds to a name being "not free in" some object-level term). The language also contains a non-deterministic branching operator, $e_1 \parallel e_2$, so that fair proof-search can be easily encoded. Without this, the programmer must implement search primitives themselves for every system. It is more convenient to have search features as a language primitive.

3.3 Static semantics

In this section we present a type system for rejecting ill-formed αML programs. For simplicity's sake we use a simple, monomorphic type system.

3.3.1 Types

The grammar of αML types T is as follows.

T	::=	E	(equality type)
		D	(data sort)
		`prop`	(type of semi-decidable propositions)
		$T \to T$	(function type)
		`(`T`,...,`T`)`	(tuple type)
		`unit`	(unit type).

The most important feature of the types grammar is that it is stratified—the grammar of equality types defined in Definition 2.1.1 is included as a carefully delimited subset of the set of αML types.

Why the two levels? We must make a definite distinction between those types which have a decidable notion of equality between their values (the equality types) and those that do not. This is crucial to obtain certain desirable properties for the operational sematics of αML. We can ensure during type-checking that every equality constraint $v = v'$ in the program is between two values which both have the same equality type E, and reject the program otherwise. Thus, if a program passes the type-checker then we can guarantee that all of the constraint problems that we must solve in order to evaluate the program are decidable, which is necessary for the Progress result (see Chapter 6 for a discussion of constraint solving).

The most obvious example of a type that does not have this decidable notion of equality is any higher type $T \to T'$—see, for example, (Dowek, 2001). Note that equality at function types is undecidable even if the argument and result types are both equality types. Thus, if we were to have a single-level type grammar, it would be much more difficult to statically rule out programs involving undecidable equality constraints.

The type `prop` is unusual—it is inhabited by expressions which perform computations over schematic formulae and α-inductive definitions, which were defined in Chapter 2. In most cases, this will involve some kind of proof-search procedure over a set of schematic rules.

Historical note 3.3.1 (Equality types and `prop`). It turns out that equality between values of type `prop` is decidable because, as we shall see, there is only one such value (however, equality between α-inductive definitions *is* undecidable). Therefore it does fit the criteria for inclusion as an equality type. However we took the design decision not to make `prop` an equality type so that equality types are reserved for representing abstract syntax.

In an early version of the MLSOS language we actually used the `unit` type instead of `prop` to denote the result of a proof-search computation but discovered that the unusual use of the `unit` type was confusing for audiences. ◇

Data sorts D are strictly more general than the *nominal* data sorts S introduced in Definition 2.1.1. Whereas a nominal data sort S may only have constructors $K_S : E \to S$ (note that the argument type of the constructor must be an equality type), general data sorts may have constructors $K_D : T \to D$, where the argument type may be of any type T. Hence we retain the full power of the ML type system, where it is possible to write datatype definitions such as

```
datatype foo = IntFun of int → int | BoolFun of bool → bool
```

to pass higher-order values around as tagged data. Equality types E are included in the grammar of types T, so nominal data sorts are a special case of general data sorts in αML. However, since equality at higher types is undecidable, data sorts D cannot be equality types. Therefore they must inhabit the "upper level" of the grammar of types.

Thanks to the way that we have set things up, we get the pleasing result that datatype declarations in αML are a straightforward extension of nominal signatures (Definition 2.1.1).

Definition 3.3.2 (Datatype declarations). An αML datatype declaration Σ consists of:

- a finite set $\mathbb{N}_\Sigma$ of *name sorts*, ranged over by N;
- a finite set $\mathbb{D}_\Sigma$ of *data sorts*, disjoint from $\mathbb{N}_\Sigma$ and ranged over by D;
- a subset $\mathbb{S}_\Sigma \subseteq \mathbb{D}_\Sigma$ of *nominal data sorts*, ranged over by S; and
- a finite set $\mathbb{C}_\Sigma$ of *constructors* $K : T \to D$, with the proviso that if $D \in \mathbb{S}_\Sigma$ then T is actually an equality type E.

Thus, if S is a nominal data sort of Σ then we write

$$\texttt{datatype}\ S \ \texttt{=}_\Sigma\ K_1\ \texttt{of}\ E_1 \ \texttt{|}\ \cdots\ \texttt{|}\ K_n\ \texttt{of}\ E_n$$

to mean that $\{K_1, \ldots, K_n\}$ is the set of all and only constructors in Σ whose result is S, and whose argument types are $E_1, \ldots, E_n$ respectively. This mimics the syntax of an ML datatype declaration. ◇

3.3.2 Inhabitation checking

In this section we describe an important subset of αML datatype declarations, namely those which do not contain any equality types which are not inhabited by some (finite) ground tree. For example, we could declare a datatype

$$\texttt{datatype foo = } K\ \texttt{of foo}$$

which is clearly not inhabited by any finite ground tree because there is no base case for the recursion. Suppose one were to write a program $\exists x : \texttt{foo}. e$ that generated a variable x of type `foo`. Any result of evaluating e that mentioned x would be meaningless as the variable would stand for a ground tree that could not exist.

In this section we demonstrate that it is possible to compute the set of nominal data sorts $S \in \mathbb{S}_\Sigma$ which are inhabited given only the definition of Σ. This will allow us to statically reject programs that might try to generate meaningless existential variables over empty types.

Assuming a datatype declaration Σ, we let $\overline{S}$ range over $\mathcal{P}(\mathbb{S}_\Sigma)$. Note that since $\mathbb{S}_\Sigma$ is finite it follows that $\mathcal{P}(\mathbb{S}_\Sigma)$ is necessarily also finite. We will write $\mathit{ndtys}(E)$ for the set of all nominal data sorts S appearing in the equality type E. We now define a function $\psi_\Sigma : \mathcal{P}(\mathbb{S}_\Sigma) \to \mathcal{P}(\mathbb{S}_\Sigma)$ by the equation

$$\psi_\Sigma(\overline{S}) \ \triangleq\ \overline{S} \cup \{S \mid \exists K, E.\ (K : E \to S) \in \Sigma \wedge \mathit{ndtys}(E) \subseteq \overline{S}\}. \tag{3.1}$$

This function takes a set $\overline{S}$ and adds to it the set of new nominal data sorts which can be deduced to be inhabited by using the fact that the nominal data sorts in $\overline{S}$ are inhabited along with some constructor K from the datatype declaration. By the definition of ψ_Σ it follows that any finite set $\overline{S} \subseteq \mathbb{S}_\Sigma$ is a postfixed point of ψ_Σ.

Lemma 3.3.3 (Monotonicity). *If $\overline{S} \subseteq \overline{S}'$ then $\psi_\Sigma(\overline{S}) \subseteq \psi_\Sigma(\overline{S}')$.*

Proof. Assume that $\overline{S} \subseteq \overline{S}'$. Then, we get that

$$\begin{array}{rcl} \psi_\Sigma(\overline{S}) & = & \overline{S} \cup \{S \mid \exists K, E.\ (K : E \to S) \in \Sigma \wedge ndtys(E) \subseteq \overline{S}\} \\ & \subseteq & S' \cup \{S \mid \exists K, E.\ (K : E \to S) \in \Sigma \wedge ndtys(E) \subseteq \overline{S}'\} \\ & = & \psi_\Sigma(S'). \end{array}$$

Thus we have that $\psi_\Sigma(\overline{S}) \subseteq \psi_\Sigma(\overline{S}')$, as required. □

Since ψ_Σ is monotone, by Tarski's fixed point theorem (Tarski, 1955) it has a least fixed point $\mathbb{I}_\Sigma$, and we write $\psi^n_\Sigma(\overline{S})$ to stand for the n-fold application of ψ_Σ to $\overline{S}$. It follows that $\mathbb{I}_\Sigma$ can be constructed as the limit of a countable chain $\bigcup_{n \geq 0} \psi^n_\Sigma(\emptyset)$. Furthermore, since $\mathcal{P}(\mathbb{S}_\Sigma)$ is finite it follows that $\mathbb{I}_\Sigma$ must also be finite, and therefore the computation of the fixed point always terminates. This means that we can compute $\mathbb{I}_\Sigma$ for any Σ.

Intuitively, $\mathbb{I}_\Sigma$ corresponds to the set of all nominal data sorts $S \in \mathbb{S}_\Sigma$ such that S is inhabited, i.e. where $\alpha\text{-}Tree_\Sigma(S) \neq \emptyset$. We now show that the process of computing $\mathbb{I}_\Sigma$ mirrors this intuition correctly, and actually implements a check for type inhabitation.

Lemma 3.3.4. *If $\alpha\text{-}Tree_\Sigma(S) \neq \emptyset$ for all $S \in ndtys(E)$ then $\alpha\text{-}Tree_\Sigma(E) \neq \emptyset$.*

Proof. By induction on the structure of the equality type E. □

Lemma 3.3.5. *For any equality type E:*

$$(\exists n.\ ndtys(E) \subseteq \psi^n_\Sigma(\emptyset)) \iff \alpha\text{-}Tree_\Sigma(E) \neq \emptyset.$$

Proof. The proof is by induction on n, using Lemma 3.3.4. □

Since we know that the set $\mathbb{I}_\Sigma$ is always computable, the following result states that we can decide whether a given equality type E is inhabited or not, under a particular datatype declaration.

Theorem 3.3.6 (Correctness of inhabitation checking). *For any datatype declaration Σ and any equality type E, $\alpha\text{-}Tree_\Sigma(E)$ is non-empty iff $ndtys(E) \subseteq \mathbb{I}_\Sigma$.*

Proof. From the definition of $\mathbb{I}_\Sigma$ we know that $ndtys(E) \subseteq \mathbb{I}_\Sigma$ iff there exists some n such that $ndtys(E) \subseteq \psi^n_\Sigma(\emptyset)$. By Lemma 3.3.5 this is equivalent to $\alpha\text{-}Tree_\Sigma(E) \neq \emptyset$, as required. □

Henceforth we will write $\Sigma \vdash E\ inhab$ as a shorthand for $ndtys(E) \subseteq \mathbb{I}_\Sigma$, and we will assume that our nominal signatures all satisfy this criterion.

3.3.3 Typing judgement

In this section we define the αML typing judgement. In order to assign a type to an expression, however, we first need assumptions on the free variables that appear in the expression.

Definition 3.3.7 (Type environments). A *type environment* Γ is a finite partial function from variables x to types T. We write $dom(\Gamma)$ for the set of variables x for which Γ provides a mapping. Due to the nature of the grammar of types presented above, the environments Δ introduced in Chapter 2 (for assigning equality types to variables) are a subset of the type environments Γ presented here, which map variables to arbitrary types. ◇

$$\frac{x \in dom(\Gamma) \qquad \Gamma(x) = T}{\Gamma \vdash x : T} \qquad \frac{\Gamma \vdash v : T \qquad (K : T \to D) \in \Sigma}{\Gamma \vdash K\,v : D} \qquad \frac{}{\Gamma \vdash \texttt{()} : \texttt{unit}}$$

$$\frac{\Gamma \vdash v_1 : T_1 \qquad \cdots \qquad \Gamma \vdash v_n : T_n}{\Gamma \vdash \texttt{(}v_1\texttt{,} \ldots \texttt{,}v_n\texttt{)} : T_1 \ast \cdots \ast T_n} \qquad \frac{f, x \notin dom(\Gamma) \qquad \Gamma, f : T \to T', x : T \vdash e : T'}{\Gamma \vdash \texttt{fun}\, f\,(x : T) : T' = e : T \to T'}$$

$$\frac{}{\Gamma \vdash \top : \texttt{prop}} \qquad \frac{\Gamma \vdash v : N \qquad \Gamma \vdash v' : E}{\Gamma \vdash \texttt{<}v\texttt{>}v' : \texttt{[}N\texttt{]}\,E} \qquad \frac{\Gamma \vdash v : E \qquad \Gamma \vdash v' : E}{\Gamma \vdash v = v' : \texttt{prop}} \qquad \frac{\Gamma \vdash v : N \qquad \Gamma \vdash v' : E}{\Gamma \vdash v \,\texttt{\#}\, v' : \texttt{prop}}$$

$$\frac{\Gamma \vdash e : T \qquad x \notin dom(\Gamma) \qquad \Gamma, x : T \vdash e' : T'}{\Gamma \vdash \texttt{let}\, x = e \,\texttt{in}\, e' : T'} \qquad \frac{\Gamma \vdash v : T \to T' \qquad \Gamma \vdash v' : T}{\Gamma \vdash v\, v' : T'}$$

$$\frac{\begin{array}{c}\Gamma \vdash v : D \qquad D = K_1 \,\texttt{of}\, T_1 \mid \cdots \mid K_n \,\texttt{of}\, T_n \\ x_1 \neq \ldots \neq x_n \notin dom(\Gamma) \qquad \Gamma, x_1 : T_1 \vdash e_1 : T \qquad \cdots \qquad \Gamma, x_n : T_n \vdash e_n : T\end{array}}{\Gamma \vdash \texttt{case}\, v \,\texttt{of}\, K_1\, x_1 \texttt{->} e_1 \mid \cdots \mid K_n\, x_n \texttt{->} e_n : T}$$

$$\frac{\Gamma \vdash v : T_1 \ast \cdots \ast T_n \qquad i \in \{1, \ldots, n\}}{\Gamma \vdash v.i : T_i} \qquad \frac{\Gamma \vdash e : T \qquad \Gamma \vdash e' : T}{\Gamma \vdash e \parallel e' : T}$$

$$\frac{x \notin dom(\Gamma) \qquad \Sigma \vdash E\ inhab \qquad \Gamma, x : E \vdash e : T}{\Gamma \vdash \exists x : E.\, e : T}$$

Figure 3.2: Typing relation for αML values, constraints and expressions

The rules presented in Figure 3.2 define a typing judgement $\Gamma \vdash e : T$, which assigns a type T to an expression e under assumptions Γ. This set of rules extends the usual (monomorphic) typing rules for a functional programming language with tuples, sum types and recursive functions. Of the standard typing rules, the most interesting are the rules for constructed data values $K\,v$ and `case` expressions. Thanks to the way we set up the relationship between general data sorts D and *nominal* data sorts S in Definition 3.3.2, this rule also handles the case when the result type of the constructor is a nominal data sort. The syntax for `case` expressions requires that the pattern match is exhaustive, that there is only one choice per data constructor and that the only functionality of a `case` expression is to remove a single outermost data constructor. For example, we cannot deconstruct the term $K\,\texttt{(}t_1\texttt{,}\ t_2\texttt{)}$ into its components t_1 and t_2 in αML using a single pattern-match such as

$$\texttt{case}\, v \,\texttt{of}\ \cdots \mid K\,\texttt{(}x_1\texttt{,}\ x_2\texttt{)} \texttt{->} \cdots \mid \cdots$$

Instead, we must evaluate a case expression to remove the outer data constructor and then use projections to get at the contents of the pair, i.e.:

$$\texttt{case}\, v \,\texttt{of}\ \cdots \mid K\,x \texttt{->} \texttt{let}\, x_1 = x.1 \,\texttt{in}\, \texttt{let}\, x_2 = x.2 \,\texttt{in}\ \cdots \mid \cdots$$

The rules which refer to the novel features of αML are all worthy of comment. The rule for an abstraction $\texttt{<}v\texttt{>}v'$ requires that the value v in abstraction position is of some name sort N. By inspection of the rules we see that this only possible if v is a variable x such that $x \in dom(\Gamma)$ and $\Gamma(x) = N$ because the sets $\mathbb{D}_\Sigma$ and $\mathbb{N}_\Sigma$ are disjoint, so we cannot produce a value of name sort

using a constructor K. Furthermore, the body v' of the abstraction (which models the lexical scope of v) must be assigned an equality type E. This is important for ensuring that constraints involving abstractions are always decidable. The type system of FreshML (Shinwell et al., 2003; Shinwell, 2005) has a different typing rule, equivalent to

$$\frac{\Gamma \vdash v : N \qquad \Gamma \vdash v' : T}{\Gamma \vdash \texttt{<}v\texttt{>}v' : \texttt{[}N\texttt{]}\,T}$$

where the body of the abstraction may be of any type, including function types. This is possible because deconstruction of abstractions is done using generative unbinding and run-time swapping as opposed to constraint solving (Pitts and Shinwell, 2008).

Fresh Objective Caml (Shinwell, 2006) actually goes even further and allows more complex types than N to appear in the abstraction position in the above rule. One can bind a whole list of names in one go by writing a (typeable) expression such as $\texttt{<[}n_1;\ldots;n_k\texttt{]>}e$, but we will not discuss this further because single name-binding is adequate for our purposes.

There are three rules for assigning the `prop` type to an expression. The first of these is for the expression $\top$, which is the only value of type `prop`. This value indicates successful completion of a proof-search computation over an α-inductive definition. There is no corresponding value to signal failure.

The rule for an equality constraint $v = v'$ requires that v and v' both have the same equality type E. As mentioned above, this ensures that the corresponding constraint problem is decidable. The rule for a freshness constraint $v \;\#\; v'$ is similar to the abstraction rule described above and implies that the value v must be a variable x of name sort. The following lemma relates the well-formedness judgement $\Delta \vdash c\ ok$ on constraints (defined in Figure 2.3) to the αML typing judgement.

Lemma 3.3.8. *For all Δ and c, $\Delta \vdash c\ ok$ iff $\Delta \vdash c : \texttt{prop}$.* □

Historical note 3.3.9 (`prop` vs. ans). In MLSOS (Lakin and Pitts, 2008) the equivalent of the "`prop`" type was called "`ans`", to abbreviate "answer". The name was changed in later versions of the language so as to be more suggestive of the kinds of computation that inhabit the type, namely those which determine the validity of some (semi-decidable) proposition. ◇

In the rule for existentials, the side-condition $x \notin dom(\Gamma)$ can always be satisfied because the variable x is bound and can be freely α-renamed at the meta-level. The newly-generated variable must be of an equality type E, which means that programs can only generate existential variables to stand for unknown α-trees and not for unknown functions, for example. The body of the expression (where the variable x is bound) may be of any type.

The inhabitation side-condition of the existential typing rule rules out programs which might try to perform existential quantification over a type that is not inhabited. By Theorem 3.3.6 this check decides whether there exist any ground trees $g \in Tree_\Sigma(E)$. If this is not the case, it would be unsound to create variables to stand for ground trees that do not exist, as described in Section 3.3.2, so the program is rejected at compile-time. Finally, the only requirement for a branching expression is that the two sides of the branch have the same type T.

The syntax of αML requires the programmer to explicitly provide the intended types of recursive function values and existentially-quantified variables. By providing the compiler with sufficient type information at these points the task of type inference is reduced to typechecking (see Theorem 3.3.10). Although type annotations make the program more verbose, they have the advantage of forcing the programmer to think long and hard about their code and provide a limited form of documentation within the syntax of the program itself.

Theorem 3.3.10 (Properties of the αML type system).

- ***Weakening:*** *If* $\Gamma \vdash e : T$ *and* $\Gamma' \supseteq \Gamma$ *then* $\Gamma' \vdash e : T$.
- ***Uniqueness:*** *If* $\Gamma \vdash e : T$ *and* $\Gamma \vdash e : T'$ *then* $T = T'$.
- ***Decidability of typechecking:*** *Given* Γ, *e and T, it is decidable whether* $\Gamma \vdash e : T$ *holds.*
- ***Decidability of typeability:*** *Given* Γ *and e, it is decidable whether there exists a type T such that* $\Gamma \vdash e : T$ *holds.*

Proof. Weakening and uniqueness are proved by straightforward inductions over the typing rules from Figure 3.2. The presence of explicit type annotations means that typechecking and typeability are equivalent, and can be decided by applying the (syntax-directed) typing rules until the derivation is completed or the search procedure fails. □

3.4 α-tree constraint problems

Constraint solving is fundamental to the operational semantics of αML (see Section 3.5). In this section we introduce α-tree constraint problems and satisfaction of these, and discuss the computational complexity of α-tree constraint solving.

Definition 3.4.1 (α-tree constraint problems). A formula $\varphi \in Form_\Sigma$ is an *α-tree constraint problem* if it is of the form

$$\exists x_1 : E_1. \cdots \exists x_m : E_m. \, (c_1 \mathbin{\&} \cdots \mathbin{\&} c_n), \tag{3.2}$$

where the atomic constraints c_i are each either an equality ($p = p'$) or a freshness ($x \# p$). ◇

We will typically write $\exists\Delta(\overline{c})$ to abbreviate the constraint problem from (3.2), where Δ is the type environment $\{x_1 : E_1, \ldots, x_n : E_n\}$ which assigns equality types to variables and $\overline{c}$ is the conjunction $c_1 \mathbin{\&} \cdots \mathbin{\&} c_n$.

Historical note 3.4.2 (Conjunctions vs. sets). An earlier version of the language (Lakin and Pitts, 2008) used finite sets of atomic constraints in configurations as opposed to finite conjunctions. We adopt the alternative approach here so that the "constraint" portion of configurations becomes a subset of the language of schematic formulae developed in Chapter 2. The advantage of using finite sets is the underlying mathematical theory of set structure—we will silently identify our finite conjunctions up to a structural congruence which allows us to reorder the atomic constraints within the conjunction and delete duplicate elements. The empty conjunction (i.e. the true formula $\top$) is identified with the empty set of constraints. ◇

The definition of α-tree constraint problems as a subset of schematic formulae means that they inherit certain properties of formulae:

- **Variable binding:** in an existential formula $\exists x : E. \, \varphi$ the variable x is bound in φ. Therefore, in an α-tree constraint problem $\exists\Delta(\overline{c})$, all of the variables in $dom(\Delta)$ are bound in $\overline{c}$.
- **Typing judgement:** the typing relation for constraint problems, $\Gamma \vdash \exists\Delta(\overline{c})\ ok$, holds if the typing judgement $\Gamma, \Delta \vdash c : \texttt{prop}$ holds for all $c \in \overline{c}$ and if $dom(\Gamma) \cap dom(\Delta) = \emptyset$. Note that the side condition on the domains of Γ and Δ can always be satisfied by α-renaming.

  This definition allows us to reason not only about constraint problems $\exists\Delta(\overline{c})$ which are closed (i.e. have no free variables), as in Figure 2.3, but also about problems where $\overline{c}$ contains some variables which are not bound by the variables in $dom(\Delta)$.

Definition 3.4.3 (Satisfiable *α*-tree constraint problems). For a constraint problem $\exists\Delta(\overline{c})$ such that $\emptyset \vdash \exists\Delta(\overline{c})$ *ok*, we say that

$$\models \exists\Delta(\overline{c}) \tag{3.3}$$

holds iff there exists a valuation $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ such that $V \models c$ for all $c \in \overline{c}$. ◇

For example, suppose that we have variables x and y of some name sort N. Then, the α-tree constraint problem

$$<x>y = <y>x$$

is satisfied by any valuation V such that $V(x) = V(y)$, as both sides of the equality constraint are instantiated to the same α-equivalence class $[<n>n]_\alpha$. Note that that if the names were modelled using implicitly permutative names n_x and n_y then the associated constraint problem

$$<n_x>n_y = <n_y>n_x \tag{3.4}$$

is *not* satisfiable, because the two terms are ground but are not in the same α-equivalence class. (3.4) corresponds to the α-tree constraint problem

$$<x>y = <y>x \;\&\; x \# y$$

where we simulate permutative behaviour by adding appropriate freshness constraints (see Chapter 5 for more details). This constraint problem is also unsatisfiable, because the first constraint $<x>y = <y>x$ is only satisfiable by a valuation V if $V(x) = V(y)$. However, such a valuation cannot satisfy the freshness constraint $x \# y$.

Definition 3.4.4. We write NonPermSat for the decision problem $\{(\Delta, \overline{c}) \mid \models \exists\Delta(\overline{c})\}$. ◇

Theorem 3.4.5. *NonPermSat is decidable.*

Proof. NonPermSat is decidable because it is a syntactic subset of the problem of equivariant unification, which was shown to be decidable by Cheney in his thesis (Cheney, 2004b, Chapter 7). □

3.4.1 Computational complexity of NonPermSat

In this Section we consider the computational complexity of NonPermSat. We shall show that the problem is NP-complete.

Cheney has shown (Cheney, 2004b, Chapter 7) and (Cheney, 2004a) that equivariant unification is NP-complete. That proof uses certain features of equivariant unification such as concrete names, name-permutations and permutation variables, which do not exist in our constraint problems. Here we present a far simpler proof, which does not rely on these extra features. This implies that the ability to write variables in abstraction position is sufficient to cause NP-completeness. Therefore the problem considered in (Cheney, 2004b, Chapter 7) must be equivalent to the NonPermSat problem.

For an intuition as to why NonPermSat is NP-complete, consider the following constraint problem, where x, x' are variables of name sort and t, t' are variables standing for some kind of constructed data terms.

$$<x>t = <x'>t' \tag{3.5}$$

When we try to solve this constraint, there are two possibilities to consider:

- the variables x and x' are aliased, i.e. both refer to the *same* concrete name n; and

- the variables x and x' refer to two *distinct* concrete names n and n'.

We may not know which of these two possibilities applies to the particular α-tree constraint problems we are trying to solve. Therefore, a constraint solver must try them both. The number of possibilities to explore doubles when the number of nested abstractions increases by one, producing exponential behaviour.

The nominal unification algorithm presented in (Urban et al., 2004) turns out to have polynomial time complexity (Calvès and Fernández, 2008), because no branching is required to solve constraint problems. The (paraphrased) rules required for solving the nominal unification problem equivalent to (3.5) are:

$$\{<n>t = <n>t'\} \uplus \overline{c} \quad \longrightarrow_{NU} \quad \{t = t'\} \cup \overline{c} \tag{3.6}$$

$$\{<n>t = <n'>t'\} \uplus \overline{c} \quad \longrightarrow_{NU} \quad \{t = (n\,n') \cdot t', n \# t'\} \cup \overline{c} \qquad \text{if } n \neq n' \tag{3.7}$$

The left-hand sides of these rules do not overlap, and since nominal unification problems may only contain concrete names in abstraction position we can always tell which rule to apply—hence no branching is required and exponential behaviour is avoided. Similar behaviour can arise in an α-tree constraint problem if extra constraints on the bound names are added. For example, if we added an equality constraint to (3.5) to produce the problem

$$<x>t = <x'>t' \;\&\; x = x' \tag{3.8}$$

then this will only fire the rule equivalent to (3.6). If we added the freshness constraint $x \# x'$ instead, this would restrict to the rule equivalent to (3.7).

We have already shown that NonPermSat is decidable (Theorem 3.4.5). We begin the proof that NonPermSat is NP-complete by showing that the decision problem can be decided by a non-deterministic Turing machine in polynomial time.

Lemma 3.4.6. *NonPermSat is in NP.*

Proof. Consider the α-tree constraint problem $\exists\Delta(\overline{c})$. We can decompose Δ as the type environment $\Delta', x_1 : N_1, \ldots, x_n : N_n$ where none of the $x \in dom(\Delta')$ are mapped to name sorts. Thus, we isolate all the variables which are assigned name sorts by Δ.

Now, there are finitely many ways that we could impose freshness (i.e. inequality) constraints between the variables $x_1, \ldots, x_n$. Each of these corresponds to a nominal unification problem obtained by instantiating the variables $x_1, \ldots, x_n$ with concrete names $n_{x_1}, \ldots, n_{x_n}$ such that $n_{x_i} \neq n_{x_j}$ iff $x_i \# x_j$ is asserted.

Since nominal unification problems can be decided in polynomial time (Calvès and Fernández, 2008), a non-deterministic Turing machine can solve an α-tree constraint problem in polynomial time by guessing the appropriate instantiation of the variables of name sort and verifying it using nominal unification. Hence NonPermSat is in NP. □

In order to show that NonPermSat is NP-complete it only remains to show that NonPermSat is NP-hard. To this end, we use the novel observation that GRAPH 3-COLOURABILITY can be encoded as a constraint problem. GRAPH 3-COLOURABILITY is an NP-complete decision problem—see any text on complexity theory such as (Papadimitriou, 1994).

Definition 3.4.7 (GRAPH 3-COLOURABILITY). We will represent a finite graph $\mathcal{G}$ by:

- a finite set $\mathcal{E}$ of *edges* e;
- a finite set $\mathcal{V}$ of *vertices* v; and

- a source function *src* and a target function *tgt* which map every edge $e \in \mathcal{E}$ to its source and target vertices, written $src(e)$ and $tgt(e)$ respectively.

Given a finite graph $\mathcal{G}$, GRAPH 3-COLOURABILITY is the problem of whether the vertices of $\mathcal{G}$ can be assigned *colours* from the set {**red**, **green**, **blue**} such that the colour of $src(e)$ is different from the colour of $tgt(e)$ for all $e \in \mathcal{E}$. ◇

We now use GRAPH 3-COLOURABILITY to show that NonPermSat is NP-complete, by first showing that membership of a finite set can be encoded as an α-tree constraint problem. In stating this Lemma we use the abbreviation $<x_1, \ldots, x_k>(-)$ to stand for the iterated abstraction $<x_1>(\cdots <x_k>(-)\cdots)$.

Lemma 3.4.8 (Finite set membership as an α-tree constraint problem). *Fix a list of distinct variables $x, x_1, \ldots, x_k, x', x'_1, \ldots, x'_k$, all of the same name sort N. Then define $mem(x, x_1, \ldots, x_k)$ to be the α-tree constraint problem*

$$\exists x', x'_1, \ldots, x'_k : N.\,(x \,\#\, x' \;\&\; <x_1, \ldots, x_k>x = <x'_1, \ldots, x'_k>x'). \tag{3.9}$$

If V is a valuation with $dom(V) = \{x, x_1, \ldots, x_k\}$ then $V \models mem(x, x_1, \ldots, x_k)$ iff $V(x)$ is a member of the finite set $\{V(x_1), \ldots, V(x_k)\}$.

Proof. For the forward direction, suppose that $V \models mem(x, x_1, \ldots, x_k)$. This implies that $V(x) \neq V(x')$ and furthermore that $[\![<x_1, \ldots, x_k>x]\!]_V = [\![<x'_1, \ldots, x'_k>x']\!]_V$. Since x and x' are constrained to stand for distinct names, $V(x)$ may not be free in $[\![<x_1, \ldots, x_k>x]\!]_V$ and $V(x')$ may not be free in $[\![<x'_1, \ldots, x'_k>x']\!]_V$ (if they were, the equality constraint between the two abstractions would not be satisfied). Thus the equality constraint can only be satisfied if $V(x)$ is bound by one of the names $V(x_1), \ldots, V(x_k)$, i.e. if $V(x) = V(x_i)$ for some $i \in \{1, \ldots, k\}$. Hence we get that $V(x) \in \{V(x_1), \ldots, V(x_k)\}$, as required.

For the reverse direction, we assume that $V(x) \in \{V(x_1), \ldots, V(x_k)\}$. It follows that $V(x)$ is not free in $[\![<x_1, \ldots, x_k>x]\!]_V$ and we can therefore extend V with appropriate instantiations for the existentially-quantified variables $x', x'_1, \ldots, x'_k$ to ensure that both $V \models x \,\#\, x'$ and $V \models <x_1, \ldots, x_k>x = <x'_1, \ldots, x'_k>x'$ hold. Therefore we have $V \models mem(x, x_1, \ldots, x_k)$, as required. □

Theorem 3.4.9. *NonPermSat is NP-complete.*

Proof. By Lemma 3.4.6 it suffices to show that NonPermSat is NP-hard. We show this by reducing GRAPH 3-COLOURABILITY to NonPermSat.

Consider a finite graph $\mathcal{G}$ defined as in Definition 3.4.7, with edges $e_1, \ldots, e_m$ and vertices $v_1, \ldots, v_n$. For simplicity we will take the vertices of $\mathcal{G}$ to be variables $v_1, \ldots, v_n$ of some name sort N, and introduce three more variables $\boldsymbol{r}, \boldsymbol{g}, \boldsymbol{b}$ (also of sort N) to represent the three colours **red**, **green** and **blue**. Then, the formula[1] $3\text{-}col(\mathcal{G})$ is defined by

$$\exists \boldsymbol{r}, \boldsymbol{g}, \boldsymbol{b}, v_1, \ldots, v_n : N.\,(\#_{\{\boldsymbol{r},\boldsymbol{g},\boldsymbol{b}\}} \;\&\; \&_{i=1}^{n}(mem(v_i, \boldsymbol{r}, \boldsymbol{g}, \boldsymbol{b})) \;\&\; \&_{j=1}^{m}(src(e_j) \,\#\, tgt(e_j))) \tag{3.10}$$

and it just remains to show that $\models 3\text{-}col(\mathcal{G})$ precisely when $\mathcal{G}$ has a 3-colouring.

We argue that the components of $3\text{-}col(\mathcal{G})$ correspond to the intuition behind a 3-colouring. The first constraint ($\#_{\{\boldsymbol{r},\boldsymbol{g},\boldsymbol{b}\}}$) simply asserts that the three variables representing the colours must be mutually distinct. The second constraint ($\&_{i=1}^{n}(mem(v_i, \boldsymbol{r}, \boldsymbol{g}, \boldsymbol{b}))$) assigns a colour to

[1] As defined in (3.10), the formula $3\text{-}col(\mathcal{G})$ is *not* an α-tree constraint problem in the sense of Definition 3.4.1, because there are $\exists$-quantifiers inside the *mem* formulae. However it can easily be converted into an α-tree constraint problem by pulling these quantifiers out to the very front.

```
let pair = fn x -> fn y -> fn z -> z x y in
  let x1 = fn y -> pair y y in
    let x2 = fn y -> x1 (x1 y) in
      let x3 = fn y -> x2 (x2 y) in
        let x4 = fn y -> x3 (x3 y) in
          let x5 = fn y -> x4 (x4 y) in
            x5 (fn y -> y)
```

Figure 3.3: Pathological example of ML type inference

each vertex of the graph by constraining the variable representing the vertex to be equal to the variable representing the colour. Since all variables present are of the same name sort N, this corresponds to being instantiated with the (α-equivalence class of the) same concrete name n. The final constraint ($\&_{j=1}^{m}(src(e_j) \mathbin{\#} tgt(e_j))$) requires that the the vertices $src(e)$ and $tgt(e)$ must be assigned different colours, for all $e \in \mathcal{E}$. If, for some edge e, the vertices $src(e)$ and $tgt(e)$ are given the same colour (**red**, say) by the second constraint then we will have $src(e) = \boldsymbol{r}$ and $tgt(e) = \boldsymbol{r}$, which will cause the final constraint to fail because $\boldsymbol{r} \mathbin{\#} \boldsymbol{r}$ is not satisfiable.

Hence *3-col*$(\mathcal{G})$ is satisfiable iff there is some way to assign the colours **red**, **green**, **blue** to the vertices of $\mathcal{G}$ such that no two vertices joined by an edge are assigned the same colour. This corresponds precisely to the notion of $\mathcal{G}$ having a 3-colouring, as required. □

The fact that NonPermSat is NP-complete does not preclude the existence of algorithms which perform acceptably on the kinds of problem that would arise in practice during the execution of αML programs. For example, ML type inference has a doubly exponential worst case, which means that there exist terms of size n whose types are of size 2^{2^n}. Figure 3.3 presents one such pathological example whose principal type scheme is hundreds of pages long, taken from (Mairson, 1990, Appendix A). This expensive corner case does not mean that the ML type inference algorithm is not useful in practice, and a similar argument exists for NonPermSat. The examples which produce large amounts of branching tend to be fairly artificial, and in particular if the depth of nested abstractions is fairly small then the amount of branching is manageable. We cannot think of any realistic examples of inductive definitions which require more than two levels of nested abstractions.

Remark 3.4.10 (NonPermSat and equivariant unification). Since the problems of equivariant unification and NonPermSat are both NP-complete, it follows that each can be reduced to the other in polynomial time. Therefore, *the two problems are equivalent*.

This implies that the additional features of equivariant unification, such as permutation variables and swappings within swappings, are not strictly necessary. It also lends weight to our claim that our constraint problem is a more comprehensible alternative to equivariant unification. ◇

In Chapter 6 we outline an algorithm which can solve α-tree constraint problems and which has been used in the implementation of αML.

3.5 Operational semantics

We now define the operational semantics of αML and prove some very basic properties of it. The operational semantics will be presented as a non-deterministic, small-step transition rela-

tion between abstract machine configurations. Non-determinism is included to encode proof-search behaviour elegantly within the programming language, and we choose a small-step transition relation because the interplay between non-determinism and the impure features of the language works out more cleanly.

The definition of the operational semantics presented here is somewhat high-level in that it underspecifies certain aspects of the run-time behaviour of αML programs, in particular with regard to the implementation of branching computations and the treatment of failed (or "stuck") computations. These are implementation details, discussion of which we defer to Chapter 7.

3.5.1 Value substitutions

There is just one notion of substitution in αML—that of substituting a value for a variable throughout an expression.

Definition 3.5.1 (Value substitutions). A *value substitution* $\sigma \in Sub_\Sigma(\Gamma, \Gamma')$ is a finite partial function which maps each variable $x \in dom(\Gamma)$ to a value $\sigma(x) \in Val_\Sigma$ such that $\Gamma' \vdash \sigma(x) : \Gamma(x)$. We write $dom(\sigma)$ for its domain of definition, and $(-)[\sigma]$ for the simultaneous capture-avoiding substitution of $\sigma(x)$ for all free occurrences of x in $(-)$, for all $x \in dom(\sigma)$.

We write $[v_1/x_1, \ldots, v_n/x_n]$ for the substitution σ such that $dom(\sigma) = \{x_1, \ldots, x_n\}$ and $\sigma(x_i) = v_i$ for all $i \in \{1, \ldots, n\}$. $\diamond$

Note that substitution is *simultaneous*, so $(x, y)[y/x, t/y]$ becomes (y, t) not (t, t), for example. Furthermore, substitution is *capture-avoiding*, so $(\exists x : E. e)[t/x]$ is just $\exists x : E. e$. We now show that the αML typing judgements are preserved by this notion of substitution.

Lemma 3.5.2. *If* $\Gamma \vdash e : T$ *and* $\sigma \in Sub_\Sigma(\Gamma, \Gamma')$ *then* $\Gamma' \vdash e[\sigma] : T$.

Proof. By a lengthy induction over the typing rules from Figure 3.2. $\square$

3.5.2 Abstract machine configurations

Configurations of the αML abstract machine need to store the following information.

- the current expression being evaluated;
- a continuation that tells us what to do with the result of evaluating the current expression;
- the set of existentially-quantified variables that have been generated, along with their (equality) types; and
- a record of the α-tree equality and freshness constraints that have been processed.

The first two of these are standard for any functional programming language, whereas the latter two correspond to the two impure features of the αML language: the generation of new variables and the solving of constraints.

We can handle newly-generated existential variables easily by keeping a type environment Δ in our configurations. This records not only the set of variables that have already been chosen but also their equality types. This type information is required for constraint solving. Constraints are handled using techniques developed in the field of *constraint logic programming* (CLP), which extends traditional logic programming languages such as Prolog with the ability to solve constraints over an arbitrary constraint domain. See (Jaffar et al., 1998) for a survey of this research area. In presenting the operational semantics of CLP programs, that paper

uses configurations of the form $\langle G \mid c \rangle$, where G is the "goal" that remains to be proved and c is a record of the constraints that have been accumulated thus far. We use this approach to constraint solving by recording a conjunction of atomic constraints within abstract machine configurations. This conjunction of atomic constraints contains all constraints that have been encountered up to the current point in the evaluation of the program. When taken with the type environment from the configuration, these produce an α-tree constraint problem.

The main benefit of our constraint-based approach is that the details of constraint solving are abstracted away and do not play a major role in the operational semantics of the programming language. Furthermore, the grammar of constraints is irrelevant to the operational semantics—any constraint encountered during evaluation is simply passed on to the constraint solver for processing. The result returned by the constraint solver then determines whether or not a transition can be made in the operational semantics.

We will combine this constraint-based approach with evaluation contexts in the style of Felleisen (Felleisen and Friedman, 1986), which we will use to represent the continuation corresponding to the remainder of the current computation. We will formalise this using frame stacks (Pitts, 2002).

Definition 3.5.3 (Frame stacks). Frame stacks, F, are defined by the following grammar.

$$\begin{array}{rcll} F & ::= & \mathtt{Id} & \text{(empty stack)} \\ & & F \circ (x.\,e) & \text{(non-empty stack).} \end{array}$$

In the case of a non-empty stack, the variable x is bound in the expression e (but not in the rest of the stack F) and we identify frame stacks up to α-conversion of these bound variables. ◇

The empty stack `Id` means that we have nothing else to do with the result from the expression currently being evaluated, and the non-empty stack $F \circ (x.\,e)$ binds the result v of the current exprssion to x in the body of e and continues by evaluating $e[v/x]$ with the smaller frame stack F.

The typing judgement for frame stacks has the form $\Gamma \vdash F : T \to T'$, which means that the stack accepts a value of type T and that the overall result of the computation has type T'. This judgement is defined by the following two rules.

$$\frac{}{\Gamma \vdash \mathtt{Id} : T \to T} \qquad \frac{\Gamma \vdash F : T'' \to T' \qquad \Gamma, x : T \vdash e : T'' \qquad x \notin dom(\Gamma)}{\Gamma \vdash F \circ (x.\,e) : T \to T'}$$

The rule for the empty stack reflects the fact that `Id` does nothing with the value provided to it. In the case of the non-empty stack, if $F \circ (x.\,e)$ accepts a value of type T and e is assigned type T'' (in the environment extended such that $x : T$), then F should accept a value of type T'' and produce a result of type T'. The side-condition that x may not appear in $dom(\Gamma)$ can always be satsfied by α-renaming the bound variable in the frame stack.

We can now present the configurations of the αML abstract machine. In the interests of providing a clean presentation of the operational semantics which is modular with regard to the impure features of the language, we will distinguish between *pure* and *impure* configurations.

Definition 3.5.4 (Pure configurations). These have the form $\langle F, e \rangle$, where F is a frame stack and e is the expression currently being evaluated. ◇

Definition 3.5.5 (Impure configurations). These are of the form $\exists \Delta(\overline{c}; F; e)$, where Δ is a type environment mapping variables to equality types, $\overline{c}$ is a finite conjunction of atomic constraints, F is a frame stack and e is the expression currently being evaluated. The variables in $dom(\Delta)$ are bound in the rest of the configuration—we identify impure configurations up to α-conversion of these bound variables. ◇

Pure configurations only contain information relevant to the evaluation of traditional functional programs, which do not contain any of the novel features of αML, whereas impure configurations also contain information on the side-effecting features.

We define a typing relation $\Gamma \vdash \exists\Delta(\overline{c}; F; e) : T$ for impure configurations, which holds if $dom(\Gamma) \cap dom(\Delta) = \emptyset$ and, for some type T', the following all hold:

- $\Gamma, \Delta \vdash e : T'$;
- $\Gamma, \Delta \vdash F : T' \to T$; and
- $\Gamma, \Delta \vdash c : \texttt{prop}$, for all $c \in \overline{c}$.

Since we identify configurations up to α-conversion of the variables in $dom(\Delta)$, it follows that the side condition $dom(\Gamma) \cap dom(\Delta) = \emptyset$ can always be satisfied. The rest of this definition is as one would expect—all atomic constraints in the conjunction $\overline{c}$ must be well-formed, and if the overall result of the computation is to be of type T then F should accept values of some type T' and produce results of type T, where T' is also the type of e.

3.5.3 Transition rules

We now turn to the rules which define the operational semantics of αML. Figure 3.4 presents small-step rules for two transition relations:

- $\langle F, e \rangle \to_P \langle F', e' \rangle$ between two pure configurations; and
- $\exists\Delta(\overline{c}; F; e) \longrightarrow \exists\Delta'(\overline{c}'; F'; e')$ between two impure configurations.

The $\to_P$ relation captures the behaviour of a standard eager functional programming language with `case` expressions, tupling, projection and recursive functions.

Rules (P1) and (P2) are worthy of comment as they are responsible for manipulating frame stacks. Since αML expressions are required to be in A-normal form (see Section 3.2), frame stacks and `let` bindings play a vital role in guiding control flow. Rule (P2) is triggered when we must evaluate the expression $\texttt{let}\, x = e \;\texttt{in}\; e'$, and pushes a new frame $(x.\, e')$ onto the top of the stack before continuing to evaluate e. If the evaluation of e produces a value v, rule (P1) fires and continues by evaluating $e'[v/x]$, which is analogous to evaluating e' under the binding "$x \mapsto v$". This captures the intuition behind `let` bindings in ML.

Note that rule (P1) cannot fire if the frame stack is empty (`Id`). In this case there is no applicable transition rule and evaluation terminates—see Section 3.7.1 for definitions of successful termination.

Rule (P3) implements the standard notion of function application for a language with recursive functions—when a value v is applied to a function $\texttt{fun}\, f(x : T) : T' = e$, the argument v is substituted for x in e and the entire function is substituted for f in e, to permit recursive calls. Rules (P4) and (P5) define the usual destructors for constructed data and tuples in ML respectively. Rule (P4) simply picks the appropriate element of the tuple, and (P5) transitions to the appropriate arm of the `case` expression, substituting the body v of the data term $K_i\, v$ for the pattern variable x_i in the process.

It is worth noting that we do not need rules for evaluating inside an evaluation context, because of our use of A-normal form and frame stacks. The syntax of the language means that the evaluation contexts are trivial. As noted in Section 3.2, this simplifies the presentation of the operational semantics and of the proofs but does not restrict the expressivity of the language.

The $\longrightarrow$ relation between impure configurations is defined in terms of $\to_P$ via (I1), which states that an impure configuration may make a transition that does not affect the impure

Pure transitions: $\langle F, e\rangle \rightarrow_P \langle F', e'\rangle$

(P1) $\langle F \circ (x.\,e), v\rangle \rightarrow_P \langle F, e[v/x]\rangle$.

(P2) $\langle F, (\texttt{let}\, x = e \,\texttt{in}\, e')\rangle \rightarrow_P \langle F \circ (x.\,e'), e\rangle$.

(P3) $\langle F, v\,v'\rangle \rightarrow_P \langle F, e[v/f, v'/x]\rangle$
if v is $\texttt{fun}\, f\,(x : T) : T' = e$.

(P4) $\langle F, (\texttt{case}\, K_i\, v\, \texttt{of}\, K_1\, x_1 \texttt{->} e_1 \mid \cdots \mid K_n\, x_n \texttt{->} e_n)\rangle \rightarrow_P \langle F, e_i[v/x_i]\rangle$
if $i \in \{1, \ldots, n\}$.

(P5) $\langle F, (v_1, \ldots, v_n).\,i\rangle \rightarrow_P \langle F, v_i\rangle$
if $i \in \{1, \ldots, n\}$.

Impure transitions: $\exists\Delta(\overline{c}; F; e) \longrightarrow \exists\Delta'(\overline{c}'; F'; e')$

(I1) $\exists\Delta(\overline{c}; F; e) \longrightarrow \exists\Delta(\overline{c}; F'; e')$
if $\langle F, e\rangle \rightarrow_P \langle F', e'\rangle$.

(I2) $\exists\Delta(\overline{c}; F; c) \longrightarrow \exists\Delta(\overline{c} \,\&\, c; F; \mathsf{T})$
if $\models \exists\Delta(\overline{c} \,\&\, c)$.

(I3) $\exists\Delta(\overline{c}; F; \exists x : E.\, e) \longrightarrow \exists\Delta, x : E(\overline{c}; F; e)$
if $x \notin dom(\Delta)$.

(I4) $\exists\Delta(\overline{c}; F; \texttt{case}\, x\, \texttt{of}\, K_1\, x_1 \texttt{->} e_1 \mid \cdots \mid K_n\, x_n \texttt{->} e_n) \longrightarrow \exists\Delta, x_i : E_i(\overline{c} \,\&\, x = K_i\, x_i; F; e_i)$
if $i \in \{1, \ldots, n\}$ and $\texttt{datatype}\; S =_\Sigma K_1\, \texttt{of}\, E_n \mid \cdots \mid K_n\, \texttt{of}\, E_n$,
where $\Delta(x) = S$ and $\models \exists\Delta, x_i : E_i(\overline{c} \,\&\, x = K_i\, x_i)$.

(I5) $\exists\Delta(\overline{c}; F; x.\,i) \longrightarrow \exists\Delta, x_1 : E_1, \ldots, x_n : E_n(\overline{c} \,\&\, x = (x_1, \ldots, x_n); F; x_i)$
if $i \in \{1, \ldots, n\}$ and $\Delta(x) = E_1 * \cdots * E_n$.

(I6) $\exists\Delta(\overline{c}; F; e_1 \parallel e_2) \longrightarrow \exists\Delta(\overline{c}; F; e_i)$
if $i \in \{1, 2\}$.

Figure 3.4: Small-step operational semantics for αML

components of the configuration if there exists a corresponding $\rightarrow_P$-transition between pure configurations. The relationship between pure and impure reductions is explored further in Section 3.6 below.

Rule (I2) processes a constraint when it is encountered during execution of the abstract machine. The current constraint c is tested for mutual satisfiability with the pre-existing constraints in $\overline{c}$, as the α-tree constraint problem $\exists\Delta(\overline{c} \,\&\, c)$. From Theorem 3.4.5 we know that this is decidable. If the constraint problem is shown to be satisfiable then it is safe to continue: the new constraint is incorporated into the constraint environment and the expression at the top of the stack becomes T to signal successful processing of the constraint. This also ensures that types are preserved by all transition steps (see Theorem 3.7.7 below). If the constraint problem $\exists\Delta(\overline{c} \,\&\, c)$ is not satisfiable then the new constraint contradicts the existing constraints, and we can proceed no further. There is no transition rule in this case, and this particular branch of the computation is said to have "failed" (see Definition 3.7.2 below for a formal definition of failure). Despite a particular branch failing, the overall computation does not necessarily fail as

there are may be other non-deterministic branches of computation that can still make progress.

Rule (I2) makes no mention of particular kinds of constraint or how they are solved, so the operational semantics is modular with respect to the grammar of constraints and the constraint solving procedure. Not only does this make for a clean and concise presentation of the operational semantics of the language, but also it means that new flavours of constraint and alternative constraint solving algorithms could be easily added. We fixed on the simple grammar of constraints presented in Figure 3.1 as it is a small constraint grammar that allows useful programs to be written.

Another benefit of our constraint-based approach is that unifying substitutions, which appear in many logic programming languages, do not occur at all. For example, consider the equality constraint

$$x = v \tag{3.11}$$

where v is some closed value which does not contain x. A constraint such as (3.11) could only arise during the execution of a closed program if the variable x was introduced by an existential quantifier. Processing this constraint using impure rule (I2) does not eliminate the variable x—in fact, a variable generated using the $\exists x : E.\, e$ construct is never eliminated from the syntax of the program during execution using the $\longrightarrow$ rules. Instead, it remains as a "free variable" and the constraint (3.11) is remembered within the constraint environment $\overline{c}$ of the configuration. Any subsequent constraint involving x must be shown to be consistent with (3.11), otherwise that branch of the computation will fail.

Historical note 3.5.6 (Unifying substitutions). In an earlier (unpublished) version of the language, the constraint grammar was not factored out of the operational semantics in the way it is in Figure 3.4, and there were explicit rules in the operational semantics for equality and freshness constraints (and a third flavour of "name inequality" constraints). In particular, unifying substitutions were computed for every equality constraint encountered during execution. After some paraphrasing to make it look similar to the rules from Figure 3.4, the rule for processing equality constraints was

$$\exists\Delta(\overline{c}; F; v = v') \longrightarrow \exists\Delta(\overline{c}'; F[\sigma]; \top) \qquad \text{if } \mathit{unify}_{\overline{c}}(v\,, v') = (\overline{c}'\,, \sigma) \tag{3.12}$$

where $\mathit{unify}_{\overline{c}}(v\,, v')$ stood for the result of unifying v and v' in the presence of the constraints $\overline{c}$. The unification process used the nominal unification algorithm from (Urban et al., 2004) to produce a substitution σ and a set of freshness (and name inequality) constraints $\overline{c}'$, as opposed to just an augmented set of atomic constraints in the more modern αML system. The unifying substitution σ then had to be applied to the entire frame stack before evaluation could proceed, which meant that occurences of a variable x would be substituted away if it appeared in an equality constraint $x = v$ (provided that x is not free in v). Not only was this potentially inefficient but it also tied the system to a particular constraint language, and the operational semantics became somewhat verbose since many extra rules were required to handle constraints. Therefore, the published versions of our work described in (Lakin and Pitts, 2008) and (Lakin and Pitts, 2009) delegate constraints to a separate constraint solver and store instantiations of existential variables implicitly within the constraint environment, as described above. ◇

Rule (I3) is responsible for generating existential variables to stand for unknown values of equality type. The side condition that x may not appear in the domain of Δ ensures that the new variable is indeed new, i.e. has not been seen before in the evaluation of this program. This constraint is trivially satisfiable by α-renaming the bound variable in the meta-language.

Historical note 3.5.7 (Checking type inhabitation). Rule (I3) from Figure 3.4 omits the runtime check for type inhabitation described in (Lakin and Pitts, 2009). In that paper, whenever

an existential quantification was encountered by the abstract machine it would only allow the transition if the equality type in question was shown to be inhabited. Otherwise, no transition was possible and the abstract machine was stuck. This approach was taken in (Lakin and Pitts, 2009) due to space constraints, and both the dynamic and static approaches to inhabitation checking suffice to prevent unsoundness. However, from a software engineering perspective it is desirable to catch such errors as soon as possible. The use of existential quantification over non-inhabited types is presumed to be an error because it is difficult to see the benefit of such a program, so we adopt the compile-time check. We also change the definition of failure (see Definition 3.7.2) so that an existential quantification can never fail. However, the type safety result in Section 3.7 still holds because if a configuration is well-typed then we know that all existential quantifications are over inhabited equality types. ◇

Rules (I4) and (I5) are the impure counterparts of rules (P4) and (P5) for deconstructing data values and tuples respectively. These rules are necessary because the ability to generate new variables of equality types using the $\exists x{:}E.\,e$ construct means that it is possible to reduce a well-typed, closed expression and reach the situation where we must evaluate an expression such as

$$\texttt{case}\, x \,\texttt{of}\, K_1\, x_1 \mathbin{\texttt{->}} e_1 \mid \cdots \mid K_n\, x_n \mathbin{\texttt{->}} e_n$$

where the variable x stands for a potentially unknown data value. Similarly, we may need to project out of an unknown tuple. In order to ensure that well-typed configurations can always make progress (unless they have terminated or failed gracefully) we need to provide transition rules to handle these cases.

Rule (I4) causes non-determinism by *narrowing* over the unknown value x. Narrowing involves non-determistically "guessing" instantiations for unknown arguments to a function or `case` expression, so that the expression may be evaluated further. There is a considerable literature on this particular kind of non-determinism, largely centred round the functional logic programming language Curry—see Hanus (2007) for a survey.

Functional logic languages tend to make a syntactic distinction between *rigid* `case` expressions (which require all of their arguments to be sufficiently instantiated) and *flexible* ones (where unknown arguments may be non-deterministically instantiated by narrowing). Furthermore, the strategy of narrowing only when absolutely necessary lends itself to a lazy evaluation strategy (Antoy et al., 2000).

For simplicity's sake, in αML we wish to avoid residuation and concurrent execution, as well as lazy evaluation. Therefore αML uses a strict evaluation strategy. As we have seen, αML features non-trivial computational effects, but we do not want to impose a monadic programming style such as that of Haskell (Peyton-Jones and Wadler, 1993). Therefore we use a simple-minded design where the "rigid/flexible" behaviour of `case` expressions is built into the dynamics of the language rather than being user-specified. If a `case` expression is evaluated over a sufficiently instantiated data value $K\,v$ then the pure rule (P4) is used, whereas if the value is a variable x then the impure rule (I4) is selected automatically instead. This relieves the user of the burden of thinking about evaluation strategies, at the cost of some non-deterministic branching.

Rule (I4) evaluates a `case` expression over an unknown data value x (of some nominal data sort S) by performing an n-way non-deterministic branch (if there are n data constructors for S). Each of these branches represents a single possible narrowed instantiation for the variable, with a different constructor at the head of the term each time. The ith possible branch of the transition generates a new existential variable x_i of sort E_i, where $(K_i : E_i \to S) \in \Sigma$. The type information Δ stored in abstract machine configurations is necessary here, as we must know the type of the variable x to know which constructors to try. This new variable stands for the hypothetical

body of the partially-instantiated term $K_i\, x_i$—the narrowing process only instantiates the head data constructor as this is all that is required to continute evaluating the `case` expressions that appear in αML. The original variable x is then constrained to be equal to $K_i\, x_i$, using an equality constraint $x = K_i\, x_i$.

We require that the constraint problem $\exists \Delta, x_i : E_i(\overline{c}\ \&\ x = K_i\, x_i)$ be satisfiable, which means that the choice of the data constructor K_i must not conflict with any existing constraints on x within $\overline{c}$. If the constraint problem is solved, evaluation continues with the body e_i of the ith branch. We evaluate e_i as an "open" expression with a free variable x_i, which represents the unknown body of the unknown data value. This contrasts with the pure rule (P4) for "rigid" `case` expressions, where the body v of the data value is substituted for v_i throughout e_i. Hence, existing constraints on x will guide us through the `case` expression in a similar way to standard ML-style pattern-matching, albeit via non-determinism.

The rule (I5) for projection from an unknown tuple x is similar. There is no need for branching or constraint solving, however, because there is only one top level "constructor" for tuples. We simply generate n new variables of the appropriate types to stand for the hypothetical elements of the unknown tuple, and transition to the variable x_i which represents the ith component of the tuple.

Rule (I6) introduces more non-deterministic branching by simply allowing the expression $e_1 \parallel e_2$ to transition *either* to e_1 *or* to e_2. This is a prime example of the underspecified nature of our definition of the operational semantics—no search strategy is specified, and no treatment of failed computation branches is prescribed. There is no communication between the various branches of an αML computation, so the concurrency introduced by branching is somewhat benign. This means that αML is ideally suited to take advantage of multicore machines by spawning different branches onto different cores (see Section 8.1.3). The astute reader might notice that the branching operator is definable in terms of other constructions, in particular flexible `case` expressions. Discussion of this point is deferred to Historical note 4.2.3 where we not only present an encoding but also justify why branching was included as a language primitive.

Definition 3.5.8 (αML programs). An αML *program* is any closed expression e, i.e. where $\emptyset \vdash e : T$ holds for some type T. The *initial configuration* for the program e is $\exists \emptyset(\mathtt{T}; \mathtt{Id}; e)$. Hence a program is started with no pre-existing variables or constraints and the identity continuation. $\diamond$

3.6 Embedded functional programming language

The pure transition relation $\rightarrow_P$ models the operational semantics of a traditional strict functional programming language. In this section we prove that this relation exists as a subset of the impure transition relation $\longrightarrow$ of αML.

Definition 3.6.1 (Purity). An expression or frame stack is *pure* if it does not contain any subexpressions of the form $\texttt{<}v\texttt{>}v'$, T, $v = v'$, $v \,\#\, v'$, $e \parallel e'$ or $\exists x : E.\, e$. $\Diamond$

Pure expressions and frame stacks correspond to the syntax of traditional functional programming languages. By inspection of the $\rightarrow_P$ rules we can conclude that the pure transition relation preserves this property, as stated in the following Lemma.

Lemma 3.6.2. *If F and e are pure and $\langle F, e\rangle \rightarrow_P \langle F', e'\rangle$ then F' and e' are also pure.* $\square$

The following main result emphasises that the operational semantics of a standard functional language can be simulated within the impure operational semantics of αML using only pure transitions.

Theorem 3.6.3 (Embedded functional programming language). *Suppose that the typing judgement $\emptyset \vdash \exists\emptyset(\mathsf{T}; F; e) : T$ holds and that F and e are pure. Then $\exists\emptyset(\mathsf{T}; F; e) \longrightarrow \exists\Delta(\overline{c}; F'; e')$ holds iff $\Delta = \emptyset$, $\overline{c} = \mathsf{T}$, F' and e' are pure, and $\langle F, e\rangle \rightarrow_P \langle F', e'\rangle$.*

Proof. If $\Delta = \emptyset$ and $\overline{c} = \mathsf{T}$ and $\langle F, e\rangle \rightarrow_P \langle F', e'\rangle$ then $\exists\emptyset(\mathsf{T}; F; e) \longrightarrow \exists\Delta(\overline{c}; F'; e')$ holds by (I1). For the converse, if $\exists\emptyset(\mathsf{T}; F; e) \longrightarrow \exists\Delta(\overline{c}; F'; e')$ holds then the transition could not be derived using (I2), (I3) or (I6) because e is pure. Furthermore, since $\emptyset \vdash \exists\emptyset(\mathsf{T}; F; e) : T$ we know that $\emptyset \vdash e : T'$ holds for some T', which means that e must be a closed expression. This rules out (I4) and (I5). Therefore the transition $\exists\emptyset(\mathsf{T}; F; e) \longrightarrow \exists\Delta(\overline{c}; F'; e')$ must have been derived using (I1), from which it follows that $\Delta = \emptyset$, $\overline{c} = \mathsf{T}$ and $\langle F, e\rangle \rightarrow_P \langle F', e'\rangle$ all hold. It just remains to see that F' and e' are pure, which follows from Lemma 3.6.2. $\square$

This result is significant—it means that it suffices to consider just impure configurations and the impure transition relation $\longrightarrow$. Henceforth we will not distinguish between pure and impure configurations, and we shall use the term "configuration" meaning only impure configurations. In Chapter 4 we will prove a similar result for a constraint logic programming language (over α-trees) embedded within the impure operational semantics of αML.

3.7 Type safety

In this section we relate the type system from Section 3.3.3 to the operational semantics presented in Figure 3.4.

3.7.1 Success and failure

We now define notions of success and failure for αML programs. Success corresponds to normal termination of a traditional functional program, and failure corresponds to a logic program answering "`no`" when a derivation cannot be found for some query. These definitions will feature in the statement of the type safety theorems in the next section and in the results from Chapter 4 and Chapter 5.

Definition 3.7.1 (Success). A configuration *has succeeded* if it is of the form $\exists\Delta(\overline{c}; \texttt{Id}; v)$, where $\models \exists\Delta(\overline{c})$ holds. A configuration *may succeed*, written $\exists\Delta(\overline{c}; F; e)\downarrow$, if there exists a finite sequence of $\longrightarrow$-reductions to a configuration that has succeeded:

$$\exists\Delta(\overline{c}; F; e) \longrightarrow \cdots \longrightarrow \exists\Delta'(\overline{c}'; \texttt{Id}; v)$$

with $\models \exists\Delta'(\overline{c}')$. We write $\exists\Delta(\overline{c}; F; \overline{c})\downarrow^n$ if there exists such a sequence of length less than or equal to n. ◇

Definition 3.7.2 (Failure). A configuration *has failed* if it takes one of the following forms:

- $\exists\Delta(\overline{c}; \texttt{Id}; v)$ where $\exists\Delta(\overline{c})$ is not satisfiable; or
- $\exists\Delta(\overline{c}; F; c')$ where $\exists\Delta(\overline{c} \mathbin{\&} c')$ is not satisfiable.

A configuration *must fail*, written $\exists\Delta(\overline{c}; F; e)$ *fails*, if every sequence of impure reductions is finite and leads to a configuration that has failed. We write $\exists\Delta(\overline{c}; F; e)$ *fails*^{n} if all such sequences are of length less than or equal to n. ◇

Historical note 3.7.3 (Failure at flexible `case` expressions). (Lakin and Pitts, 2009) defined an additional possiblity for failure: when one encounters a flexible `case` expression $\texttt{case}\,x\,\texttt{of}\ K_1\,x_1 \texttt{->} e_1 \mid \cdots \mid K_n\,x_n \texttt{->} e_n$ where $\exists\Delta, x_i\!:\!E_i(\overline{c} \mathbin{\&} \{x = K_i\,x_i\})$ is not satisfiable for any $i \in \{1,\ldots,n\}$. We omit this case from Definition 3.7.2 as it can never actually arise here. Any satisfying valuation V for the existing constraints $\overline{c}$ must map x to an α-tree $[K_i\,g]_\alpha$ for some $i \in \{1,\ldots,n\}$. Since the type system enforces that a `case` expression on values of sort S must cover all possible constructors for that type, then the equality constraint in at least one of the n branches (the ith branch in this case) must always be satisfiable in conjunction with $\overline{c}$. ◇

Historical note 3.7.4 (Failure at existential quantifiers). The definition of failure above is also slightly different from that in (Lakin and Pitts, 2009) in that here we do not define an existential quantification $\exists x\!:\!E.\,e$ over an uninhabited equality type E as a failure. The reason for this (discussed in Historical note 3.5.7 above) is that the type system presented here rules out such programs statically, so the corresponding dynamic check would never fail. ◇

A configuration terminates successfully if *some* branch of the computation succeeds (i.e. finds an answer) but only fails if *all* branches fail individually (i.e. no answer can be found). Finite failure will be discussed further in Section 5.6.

Lemma 3.7.5. *For a well-typed configuration $\exists\Delta(\overline{c}; F; e)$ (i.e. where $\varnothing \vdash \exists\Delta(\overline{c}; F; e) : T$ holds for some T), at most one of $\exists\Delta(\overline{c}; F; e)\downarrow$ and $\exists\Delta(\overline{c}; F; e)$ fails is derivable.*

Proof. It follows from Definition 3.7.1 and Definition 3.7.2 that a configuration cannot both succeed and fail. However, it *is* possible that a configuration neither succeeds nor fails, because αML allows us to write divergent programs. □

3.7.2 Safety theorems

We now prove some important properties of the operational semantics with regard to run-time type safety. The first property of interest is that satisfaction of the constraint problem in an impure configuration is preserved over $\longrightarrow$ transitions. It is important to know that the $\longrightarrow$ rules cannot turn an unsatisfiable configuration into a satisfiable one, or vice versa.

Theorem 3.7.6 (Preservation of satisfaction). *If* $\emptyset \vdash \exists\Delta(\overline{c}; F; e) : T$ *and*

$$\exists\Delta(\overline{c}; F; e) \longrightarrow \exists\Delta'(\overline{c}'; F'; e') \tag{3.13}$$

then $\models \exists\Delta(\overline{c})$ *iff* $\models \exists\Delta'(\overline{c}')$.

Proof. Suppose (3.13) holds. By inspection of the transition rules it is easy to see that $\Delta' \supseteq \Delta$ and $\overline{c}' \supseteq \overline{c}$. Hence $\models \exists\Delta'(\overline{c}')$ implies $\models \exists\Delta(\overline{c})$. To show the converse we argue by cases on the impure transition rule used to derive (3.13). The cases for (I1), (I2), (I4) and (I6) are trivial. For case (I5) we note that if $\Delta(x) = E_1 * \cdots * E_n$, $x_1, \ldots, x_n \notin dom(\Delta)$ and the $x_1, \ldots, x_n$ are mutually distinct then we can extend any valuation $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ such that $V \models \overline{c}$ to produce a new valuation $V' \in \alpha\text{-}Tree_\Sigma(\Delta, x_1 : E_1, \ldots, x_n : E_n)$ such that $V' \models \overline{c}$ & $x = (x_1, \ldots, x_n)$. This is achieved by defining $V'(x_i) = [g_i]_\alpha$ for $i \in \{1, \ldots, n\}$ (where $V(x) = [(g_1, \ldots, g_n)]_\alpha$). Finally for case (I3) the fact that $\exists\Delta(\overline{c}; F; e)$ is well-typed means that $\Sigma \vdash E$ *inhab* holds. Then, by Theorem 3.3.6 there exists some ground tree $g \in Tree_\Sigma(E)$. Therefore if $V \in \alpha\text{-}Tree_\Sigma(\Delta)$, $V \models \overline{c}$ and $x \notin dom(\Delta)$, we can extend V to produce a new valuation $V' \in \alpha\text{-}Tree_\Sigma(\Delta, x : E)$ such that $V' \models \overline{c}$ by setting $V'(x) = [g]_\alpha$. ☐

We now present the standard type safety results for a functional programming language: type preservation and progress.

Theorem 3.7.7 (Type preservation). *If* $\emptyset \vdash \exists\Delta(\overline{c}; F; e) : T$ *and* $\exists\Delta(\overline{c}; F; e) \longrightarrow \exists\Delta'(\overline{c}'; F'; e')$ *then* $\emptyset \vdash \exists\Delta'(\overline{c}'; F'; e') : T$.

Proof. By cases on the rule(s) from Figure 3.4 used to derive the reduction $\exists\Delta(\overline{c}; F; e) \longrightarrow \exists\Delta'(\overline{c}'; F'; e')$. ☐

Theorem 3.7.8 (Progress). *If* $\emptyset \vdash \exists\Delta(\overline{c}; F; e) : T$ *and* $\exists\Delta(\overline{c}; F; e)$ *has neither succeeded nor failed, then* $\exists\Delta(\overline{c}; F; e) \longrightarrow \exists\Delta'(\overline{c}'; F'; e')$ *holds for some* $\exists\Delta'(\overline{c}'; F'; e')$.

Proof. By case analysis on the possible forms of the configuration $\exists\Delta(\overline{c}; F; e)$, using the definitions of success (Definition 3.7.1) and failure (Definition 3.7.2). ☐

These results tell us that well-typed αML programs are crash-free in the sense that they do not stop making impure transitions unless they have reached a state of success or failure.

Chapter 4

α-inductive definitions in αML

"Programs must be written for people to read, and only incidentally for machines to execute."
—H. Abelson and G. Sussman

In this chapter we embed α-inductive definitions into the αML meta-language in a simple and convenient way. We then relate the operational semantics of αML to that of a simple constraint logic programming language over the constraint domain of α-trees with equality and freshness constraints. We prove soundness and completeness results for the operational behaviour of embedded α-inductive definitions in αML, and close with the translation of an example definition into αML code.

4.1 α-inductive definitions as αML recursive functions

In this section we present an embedding of α-inductive definitions into αML. As in Section 2.4 we will fix on a single relation symbol r, which we will regard as an αML variable of type $S_r \to \texttt{prop}$ for a fixed nominal data sort S_r. We assume that $\mathcal{D}$ is an α-inductive definition as defined in Definition 2.4.2, and we will identify $\mathcal{D}$ with the closed αML recursive function value $v_{\mathcal{D}}$ of type $S_r \to \texttt{prop}$, where

$$v_{\mathcal{D}} \triangleq (\texttt{fun}\, r\,(x : S_r) : \texttt{prop} = \varphi) \qquad \text{where } \mathcal{D} \text{ is } \frac{\varphi}{r\,x}. \tag{4.1}$$

This means that we must embed the formula $\varphi \in Form_\Sigma$ as an αML expression e such that

$$\{r : S_r \to \texttt{prop}, x : S_r\} \vdash e : \texttt{prop} \tag{4.2}$$

holds. Thanks to the way we have set up the syntax of both the language of schematic definitions (Chapter 2) and the αML meta-language itself (Chapter 3) this is a straightforward process:

- Patterns of type E correspond exactly to αML values of the same equality type. Therefore, the constraints which appear in schematic formulae and in αML are identical.
- The $\top$ formula also exists in the syntax of schematic formulae and αML expressions.
- The atomic formula $r\,p$ corresponds to the meta-language application of the variable r, which has type $S_r \to \texttt{prop}$, to the value p of type S_r. The intuition is that r stands for the recursive function $v_{\mathcal{D}}$.

- If the formula φ can be characterised as an αML expression, then the $\exists x{:}E.\,\varphi$ formula is easily translated.
- The $\varphi_1 \vee \varphi_2$ formula trivially becomes the αML expression $\varphi_1 \parallel \varphi_2$, assuming that its subformulae φ_1 and φ_2 can be translated.

The only formulae whose translation is non-trivial are the conjunction and "false" formulae:

- e_1 & e_2 is modelled by using `let` bindings and the eager reduction strategy of αML:

$$e_1 \mathbin{\&} e_2 \;\triangleq\; \texttt{let}\, x = e_1 \,\texttt{in}\, e_2 \qquad \text{where } x \notin FV(e_2). \tag{4.3}$$

  This gives a left-to-right sequential treatment of conjunction, as the expression e_1 is evaluated first (and its result discarded) before e_2 is evaluated.

- The false formula F can be modelled by the expression

$$\mathsf{F} \;\triangleq\; \exists x{:}N.\, x \,\#\, x \tag{4.4}$$

  where $N \in \mathbb{N}_\Sigma$ is any name sort defined in the nominal signature Σ. This expression always fails finitely because no name x can be fresh for itself.

Thus, given an inductive definition $\mathcal{D}$ (with its associated αML function $v_\mathcal{D}$) every schematic formula $\varphi \in \mathit{Form}_\Sigma$ has a straightforward encoding as an αML expression $\varphi[v_\mathcal{D}/r]$ of type `prop`. Following Definition 2.4.2, we are only really interested in those formulae whose only free variables are r and x. The following lemma states that such well-formed formulae have well-typed encodings in αML, i.e. which satisfy (4.2).

Lemma 4.1.1. *For any formula φ and definition $\mathcal{D}$, if $\Delta \vdash \varphi$ ok then $\Delta \vdash \varphi[v_\mathcal{D}/r]$*$:$`prop`.

Proof. The result follows by a straightforward induction on the typing rules for formulae from Figure 2.3. □

Therefore, for any standard form α-inductive definition $\mathcal{D}$ the corresponding recursive function value $v_\mathcal{D}$ satisfies

$$\varnothing \vdash v_\mathcal{D} : S_r \to \texttt{prop}. \tag{4.5}$$

We did not define a formal translation function between formulae and their corresponding αML expressions because the mapping is so simple. This is another advantage of the design of αML—the language of α-inductive definitions maps almost directly onto αML expressions.

4.2 Embedded constraint logic programming language

We now present a third transition relation, this time between a subset of abstract machine configurations which may be interpreted as "goal states" from constraint logic programming (CLP). We then formalise the relationship between the operational semantics of this CLP language and the operational semantics of αML. Thus we conclude that αML subsumes a CLP language over the constraint domain of α-trees in the same sense that it subsumes an eager higher-order functional programming language (Theorem 3.6.3).

Figure 4.1 defines a transition judgement $\mathcal{D} \vdash \exists\Delta(\overline{c};\vec{\varphi};\varphi) \rightsquigarrow \exists\Delta'(\overline{c}';\vec{\varphi}';\varphi')$ which we call *formula reduction*. The relation is between special kinds of configuration which have a queue $\vec{\varphi}$ of formulae instead of a frame stack and where the expression at the top of the stack must correspond to some formula φ. These are a subset of αML impure configurations as defined in

(F1)	$\mathcal{D} \vdash \exists\Delta(\overline{c}; \vec{\varphi}; r\, p) \rightsquigarrow \exists\Delta(\overline{c}; \vec{\varphi}; \varphi[p/x])$	if $v_{\mathcal{D}} = \mathtt{fun}\, r(x{:}S_r)\,{:}\,\mathtt{prop} = \varphi$.
(F2)	$\mathcal{D} \vdash \exists\Delta(\overline{c}; \vec{\varphi}; c) \rightsquigarrow \exists\Delta(\overline{c}\ \&\ c; \vec{\varphi}; \top)$	if $\models \exists\Delta(\overline{c}\ \&\ c)$.
(F3)	$\mathcal{D} \vdash \exists\Delta(\overline{c}; \vec{\varphi}, \varphi; \top) \rightsquigarrow \exists\Delta(\overline{c}; \vec{\varphi}; \varphi)$	
(F4)	$\mathcal{D} \vdash \exists\Delta(\overline{c}; \vec{\varphi}; \varphi\ \&\ \varphi') \rightsquigarrow \exists\Delta(\overline{c}; \vec{\varphi}, \varphi'; \varphi)$	
(F5)	$\mathcal{D} \vdash \exists\Delta(\overline{c}; \vec{\varphi}; \varphi_1 \vee \varphi_2) \rightsquigarrow \exists\Delta(\overline{c}; \vec{\varphi}; \varphi_i)$	if $i \in \{1, 2\}$.
(F6)	$\mathcal{D} \vdash \exists\Delta(\overline{c}; \vec{\varphi}; \exists x{:}E.\, \varphi) \rightsquigarrow \exists\Delta, x{:}E(\overline{c}; \vec{\varphi}; \varphi)$	if $x \notin dom(\Delta)$.

Figure 4.1: Formula reduction

Definition 3.5.5. The judgement is moderated by an α-inductive definition $\mathcal{D}$ because in order to evaluate an atomic formula $r\,p$ we must know the definition of the relation r. These rules give a largely standard formulation of the operational semantics of CLP (Jaffar et al., 1998, Section 3). The reader is also referred to the operational semantics of nominal logic programming (Cheney and Urban, 2008, Figure 13).

The formula reduction rules bear a striking similarity to their impure counterparts from Figure 3.4. Rule (F1) is a specialised version of the application rule (P3). The recursive function value $v_{\mathcal{D}}$ that corresponds to the α-inductive definition $\mathcal{D}$ to the left of the turnstile is used in place of r, and the pattern value p is substituted for x throughout the formula that appears in $v_{\mathcal{D}}$. The constraint rule (F2) is identical to the impure rule (I2), and rule (F3), which moves on from a true formula to process the next formula in the queue, corresponds to rule (P1) in the case where the value is of type `prop`. Rule (F4) deals with conjunction by emulating the rules that deal with `let` bindings, as this is how conjunction is encoded within αML, and the remaining rules are identical to the impure rules for handling branching and existential quantification.

We now relate formula reduction to the operational semantics of αML from Figure 3.4. We begin by defining a subset of αML frame stacks which correspond to the queues $\vec{\varphi}$ discussed in the previous section.

Definition 4.2.1 (CLP goal lists). We encode a goal list $\vec{\varphi}$ as an αML frame stack $F_{\vec{\varphi}}$, defined by recursion on the length of the goal list as follows.

$$\begin{array}{rcll} F_{\emptyset} & \triangleq & \mathtt{Id} & \\ F_{\vec{\varphi},\varphi} & \triangleq & F_{\vec{\varphi}} \circ (x.\,\varphi) & \text{where } x \notin FV(\varphi). \end{array} \qquad \Diamond$$

A frame stack corresponding to a CLP goal list is just a queue of formulae waiting to be evaluated. By Lemma 4.1.1 and Theorem 3.7.7, any formula that is successfully processed will result in the value $\top$, and because the bound variable in each stack frame must not be free in the corresponding formula this value is discarded at each step. The only information passed along is the constraints and the environment of generated variables.

Theorem 4.2.2 (Embedded CLP). *Let $\mathcal{D}$ be an inductively defined relation (in the sense of Definition 2.4.2) and suppose that $\emptyset \vdash \exists\Delta(\overline{c}; F_{\vec{\varphi}}[v_{\mathcal{D}}/r]; \varphi[v_{\mathcal{D}}/r])\,{:}\,\mathtt{prop}$. Then an impure reduction*

$$\exists\Delta(\overline{c}; F_{\vec{\varphi}}[v_{\mathcal{D}}/r]; \varphi[v_{\mathcal{D}}/r]) \longrightarrow \exists\Delta'(\overline{c}'; F; e) \tag{4.6}$$

holds iff there is a formula reduction

$$\mathcal{D} \vdash \exists\Delta(\bar{c}; \vec{\varphi}; \varphi) \rightsquigarrow \exists\Delta'(\bar{c}'; \vec{\varphi}'; \varphi') \tag{4.7}$$

for some $\vec{\varphi}'$ *and* φ'*, with* $F = F_{\vec{\varphi}'}[v_{\mathcal{D}}/r]$ *and* $e = \varphi'[v_{\mathcal{D}}/r]$.

Proof. By cases according to the structure of the formula φ: (F1) corresponds to (I1) with (P3), (F2) to (I2), (F3) to (I1) with (P1), (F4) to (I1) with (P2), (F5) to (I6), and (F6) to (I3). ☐

This result shows that formula reduction can be mimicked by a subset of the αML reduction rules. Therefore, the operational semantics of αML incorporates both a standard functional programming language *and* a constraint logic programming language (over the constraint domain of α-trees).

Historical note 4.2.3 (Branching as a defined operator). It is possible to define the branching operator in terms of flexible `case` expressions (Lakin and Pitts, 2009). If we assume that the datatype declaration Σ defines a special data sort `bool`

```
datatype bool = True of unit | False of unit
```

then any branch expression may be encoded as a flexible `case` expression over the `bool` type, as follows.

$$e_1 \parallel e_2 \triangleq \exists x\texttt{:bool.}\,\texttt{case}\ x\ \texttt{of True}\ x_1 \texttt{->} e_1\ \texttt{|} \cdots \texttt{|}\ \texttt{False}\ x_2 \texttt{->} e_2 \tag{4.8}$$

where the variables x, x_1 and x_2 are not free in either e_1 or e_2. The encoding generates a new dummy variable of type `bool` to narrow over—since no constraints exist on this variable both arms of the `case` expression will be fully evaluated, which mimics the operational behaviour of $e_1 \parallel e_2$. (Lakin and Pitts, 2009) used this encoding as a shorthand due to space limitations, but we do not adopt it here because the generation of the variable x is a side-effect that propagates beyond the expressions e_1 and e_2.

While this is not problematic from the perspective of evaluating αML programs it does complicate the development of the theory, in particular the statement and proof of Theorem 4.2.2. The difficulty arises because in the case of a disjunction formula $\varphi_1 \vee \varphi_2$, the αML translation has the additional side-effect of generating a variable to narrow over. This does not happen in the world of α-inductive definitions. If disjunction were handled using the translation from (4.8), the statement of Theorem 4.2.2 would need to account for the fact that the αML reduction of a particular formula might generate more existential variables than the corresponding formula reduction. The version of Theorem 4.2.2 stated and proved above is much cleaner because the impure elements of the configuration (variables and constraints) are precisely the same in both the $\longrightarrow$ and $\rightsquigarrow$ transition systems. Therefore we include branching as a primitive operator in the meta-language. ◇

4.3 Soundness and completeness

In this section we relate the evaluation of formulae in the αML operational semantics (or, equivalently, the formula reduction semantics) to the satisfaction of formulae as defined in Definition 2.4.5. In stating the main result of this chapter we use the following definition of the set of solutions to a configuration of the $\rightsquigarrow$ relation.

Definition 4.3.1 (Solution sets). Let $\mathcal{D}$ be an inductively defined relation which we identify with $v_{\mathcal{D}} \triangleq \texttt{fun}\, r\,(x\!:\!S_r)\!:\!\texttt{prop} = \varphi$. Given $(\Delta, \overline{c}, \vec{\varphi}, \varphi)$ such that $\Delta \vdash c\ ok$ for all $c \in \overline{c}$ and $\Delta \vdash \psi\ ok$ for all $\psi \in \vec{\varphi}, \varphi$, we will write $solns_{\mathcal{D}}(\Delta, \overline{c}, \vec{\varphi}, \varphi)$ for the *solution set*

$$solns_{\mathcal{D}}(\Delta, \overline{c}, \vec{\varphi}, \varphi) \triangleq \{\exists\Delta'(\overline{c}') \mid \mathcal{D} \vdash \exists\Delta(\overline{c}; \vec{\varphi}; \varphi) \rightsquigarrow \cdots \rightsquigarrow \exists\Delta, \Delta'(\overline{c}'; \emptyset; \top) \wedge \models \exists\Delta, \Delta'(\overline{c}')\}.$$

$\diamond$

The above definition only makes sense because the type environment Δ never gets smaller across a $\longrightarrow$ transition, and therefore also across a $\rightsquigarrow$ transition. This means that the final type environment of a reduction sequence can always be expressed by adding some (possibly empty) type environment Δ' to the original Δ. By α-renaming we may assume that the domains of Δ and Δ' are disjoint. Definition 4.3.1 also exploits the fact that the constraint problem $\exists\Delta'(\overline{c}')$, which has free variables corresponding to $dom(\Delta)$, is just a schematic formula which happens to have some free variables. This means that our existing notion of formula satisfaction can be used.

We now state the fundamental correctness result which relates the semantics of schematic formulae and the operational behaviour of αML. In doing so we interpret goal lists $\vec{\varphi}$ as implicit conjunctions of individual formulae, so that they may be treated the same as normal schematic formulae.

Theorem 4.3.2 (Logical soundness and completeness). *With $\mathcal{D}$ and $(\Delta, \overline{c}, \vec{\varphi}, \varphi)$ as introduced in Definition 4.3.1, the following hold for all $V \in \alpha\text{-}Tree_{\Sigma}(\Delta)$:*

Logical soundness: *If $\exists\Delta'(\overline{c}') \in solns_{\mathcal{D}}(\Delta, \overline{c}, \vec{\varphi}, \varphi)$ and $([\![\mathcal{D}]\!], V) \models \exists\Delta'(\overline{c}')$ then $([\![\mathcal{D}]\!], V) \models (\overline{c}\ \&\ \vec{\varphi}\ \&\ \varphi)$.*

Logical completeness: *If $([\![\mathcal{D}]\!], V) \models (\overline{c}\ \&\ \vec{\varphi}\ \&\ \varphi)$ then there is some $\exists\Delta'(\overline{c}') \in solns_{\mathcal{D}}(\Delta, \overline{c}, \vec{\varphi}, \varphi)$ such that $([\![\mathcal{D}]\!], V) \models \exists\Delta'(\overline{c}')$.* $\diamond$

Theorem 4.3.2 is the main result of this chapter, and the rest of this section is devoted to its proof. The overall structure of the proof is closely related to the corresponding proofs for constraint logic programming from (Jaffar et al., 1998). The fact that we can adopt existing CLP reasoning techniques to reason about the CLP sublanguage of αML is further evidence of the close links between the two systems, demonstrated by the result of Theorem 4.2.2. The terms "logical soundness" and "logical completeness" are borrowed from (Jaffar et al., 1998).

The valuation V mentioned in Theorem 4.3.2 provides instantiations for the variables in $dom(\Delta)$ which are already existentially quantified in the starting configuration $\exists\Delta(\overline{c}; \vec{\varphi}; \varphi)$. The bindings for variables in $dom(\Delta')$ correspond to the satisfying instantiations that are discovered during evaluation of the configuration.

The statement of Theorem 4.3.2 also makes implicit use of the close relationship between the syntax of schematic formulae and that of αML. Treating the goal list $\vec{\varphi}$ as an implicit conjunction is reasonable because the configuration can only succeed if all of the formulae in φ can be simultaneously satisfied, and then this can be combined with the accumulated atomic constraints c and the currently evaluating formula φ to produce a syntactically correct schematic formula $(\overline{c}\ \&\ \vec{\varphi}\ \&\ \varphi)$. Therefore we can then use our existing mathematical infrastructure to reason about the satisfaction of the entire contents of a configuration.

4.3.1 Proof of logical soundness

In this section we work towards a proof of the *logical soundness* half of Theorem 4.3.2. This states that if a particular constraint problem is in the solution set of a configuration under the αML,

then any valuation which satisfies the constraint problem also satisfies all of the formualae in the original configuration. Intuitively, this means that the αML operational semantics does not compute any “wrong answers”.

Definition 4.3.3 (Formula entailment and equivalence). Given formulae φ and φ' such that $\Delta \vdash \varphi$ *ok* and $\Delta \vdash \varphi'$ *ok*, we write $\mathcal{D}, \mathcal{D}' \models \forall\Delta.\, \varphi \Longrightarrow \varphi'$ to mean that, for all $V \in \alpha\text{-}Tree_\Sigma(\Delta)$, if $([\![\mathcal{D}]\!], V) \models \varphi$ then $([\![\mathcal{D}']\!], V) \models \varphi'$. We write $\mathcal{D}, \mathcal{D}' \models \forall\Delta.\, \varphi \equiv \varphi'$ for the symmetric version of this relation. If $\mathcal{D} = \mathcal{D}'$ we abbreviate these to $\mathcal{D} \models \forall\Delta.\, \varphi \Longrightarrow \varphi'$ and $\mathcal{D} \models \forall\Delta.\, \varphi \equiv \varphi'$ respectively. ◇

This notion of implication between formulae will be used in our proof of logical soundness. We begin by enumerating some straightforward properties of pattern valuation with regard to weakening and substitution.

Lemma 4.3.4. *Suppose that $\Delta \subseteq \Delta'$, $V' \in \alpha\text{-}Tree_\Sigma(\Delta')$ and that $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ is the restriction of V' to $dom(\Delta)$. Then:*

1. *if $\Delta \vdash p : E$ then ($\Delta' \vdash p : E$ and) $[\![p]\!]_V = [\![p]\!]_{V'} \in \alpha\text{-}Tree_\Sigma(E)$.*
2. *if $\Delta \vdash c : \mathtt{prop}$ then ($\Delta' \vdash c : \mathtt{prop}$ and) $V \models c \Longleftrightarrow V' \models c$.*
3. *if $\Delta, \{r : S_r \to \mathtt{prop}\} \vdash \varphi : \mathtt{prop}$ and $R \subseteq \alpha\text{-}Tree_\Sigma(S_r)$ then ($\Delta', \{r : S_r \to \mathtt{prop}\} \vdash \varphi : \mathtt{prop}$ and) $(R, V) \models \varphi \Longleftrightarrow (R, V') \models \varphi$.* □

Lemma 4.3.5. *Suppose that*

$$\Delta, \{r : S_r \to \mathtt{prop}, x : E\} \vdash \varphi : \mathtt{prop} \tag{4.9}$$

$$\Delta, \{x : E\} \vdash p' : E' \tag{4.10}$$

$$\Delta \vdash p : E \tag{4.11}$$

all hold. Then for any α-tree valuation $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ and any α-tree relation $R \subseteq \alpha\text{-}Tree_\Sigma(S_r)$:

$$[\![p'[p/x]]\!]_V = [\![p']\!]_{V[x \mapsto [\![p]\!]_V]} \tag{4.12}$$

$$(R, V) \models \varphi[p/x] \Longleftrightarrow (R, V[x \mapsto [\![p]\!]_V]) \models \varphi \tag{4.13}$$

Proof. Property (4.12) follows from the definition in Lemma 2.3.4, by induction on the structure of the pattern p'. Then property (4.13) can be proved by induction on the structure of the formula φ, using (4.12) in the case when φ is an atomic formula $r\, p'$ and using Lemma 4.3.4(1) if φ is $\exists x' : E'.\, \varphi'$. □

The following lemma is the main result needed to prove the logical soundness result. We prove that if the configuration $\exists\Delta(\overline{c}; \vec{\varphi}; \varphi)$ transitions to $\exists\Delta'(\overline{c}'; \vec{\varphi}'; \varphi')$, then any valuation which satisfies $(\overline{c}' \,\&\, \vec{\varphi}' \,\&\, \varphi')$ will also satisfy $(\overline{c} \,\&\, \vec{\varphi} \,\&\, \varphi)$. This means that the $\rightsquigarrow$ transition relation may narrow down the set of valuations which satisfy the configuration but it may not add extra satisfying valuations—it would be unsound to report satisfying valuations which do not satisfy the initial configuration.

Lemma 4.3.6. *Let $\mathcal{D}$ be an α-inductive definition in the sense of Definition 4.3.1, and treat goal lists $\vec{\varphi}$ as implicit conjunctions as before. Then if*

$$\mathcal{D} \vdash \exists\Delta(\overline{c}; \vec{\varphi}; \varphi) \rightsquigarrow \exists\Delta'(\overline{c}'; \vec{\varphi}'; \varphi') \tag{4.14}$$

holds, then $\Delta \subseteq \Delta'$ and

$$\mathcal{D} \models \forall\Delta'.\, (\overline{c}' \,\&\, \vec{\varphi}' \,\&\, \varphi') \Longrightarrow (\overline{c} \,\&\, \vec{\varphi} \,\&\, \varphi). \tag{4.15}$$

Proof. We proceed by cases on the formula reduction rule used to derive (4.14).

(F1). In this case we have $\varphi = r\,p$, for some p. Using rule (F1) and the definition of $\mathcal{D}$ we get that $\Delta' = \Delta$, $\overline{c}' = \overline{c}$, $\vec{\varphi}' = \vec{\varphi}$ and $\varphi' = \psi[p/x]$. Given some valuation $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ such that $(\llbracket\mathcal{D}\rrbracket, V) \models \overline{c} \mathbin{\&} \vec{\varphi} \mathbin{\&} \psi[p/x]$, to prove the result it suffices to show that $(\llbracket\mathcal{D}\rrbracket, V) \models r\,p$ holds. By (4.13) we get that $(\llbracket\mathcal{D}\rrbracket, V[x \mapsto \llbracket p\rrbracket_V]) \models \psi$, and using Lemma 4.3.4 (and the definition of $\llbracket\mathcal{D}\rrbracket$ as the least fixed-point of a monotone operator in Definition 2.4.7) we get that $(\llbracket\mathcal{D}\rrbracket, V) \models r\,p$ holds, as required.

(F2). Here we have $\varphi = c$, for some c. Then it follows that $\Delta' = \Delta$, $\overline{c}' = \overline{c} \mathbin{\&} c$, $\vec{\varphi}' = \vec{\varphi}$ and $\varphi' = \top$, and also that $\models \exists\Delta(\overline{c} \mathbin{\&} c)$ holds. Given an arbitrary valuation $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ such that $(\llbracket\mathcal{D}\rrbracket, V) \models (\overline{c} \mathbin{\&} c) \mathbin{\&} \vec{\varphi} \mathbin{\&} \top$, it follows trivially that $(\llbracket\mathcal{D}\rrbracket, V) \models \overline{c} \mathbin{\&} \vec{\varphi} \mathbin{\&} \varphi$ holds, as required.

(F3). In this case we know that $\vec{\varphi} = \vec{\varphi}^*, \varphi^*$ and $\varphi = \top$, for some $\vec{\varphi}^*$ and φ^*. We get that $\Delta' = \Delta$, $\overline{c}' = \overline{c}$, $\vec{\varphi}' = \vec{\varphi}^*$ and $\varphi' = \varphi^*$ all hold. Then, it is trivially the case that $(\llbracket\mathcal{D}\rrbracket, V) \models \overline{c} \mathbin{\&} \vec{\varphi}^* \mathbin{\&} \varphi^*$ implies $(\llbracket\mathcal{D}\rrbracket, V) \models \overline{c} \mathbin{\&} (\vec{\varphi}^*, \varphi^*) \mathbin{\&} \top$ holds for any $V \in \alpha\text{-}Tree_\Sigma(\Delta)$.

(F4). We get that $\varphi = \varphi_1 \mathbin{\&} \varphi_2$, for some φ_1 and φ_2. By rule (F4) we have that $\Delta' = \Delta$, $\overline{c}' = \overline{c}$, $\vec{\varphi}' = \vec{\varphi}, \varphi_2$ and $\varphi' = \varphi_1$. Then, it follows that $(\llbracket\mathcal{D}\rrbracket, V) \models \overline{c} \mathbin{\&} (\vec{\varphi}, \varphi_2) \mathbin{\&} \varphi_1$ implies $(\llbracket\mathcal{D}\rrbracket, V) \models \overline{c} \mathbin{\&} \vec{\varphi} \mathbin{\&} (\varphi_1 \mathbin{\&} \varphi_2)$, as required.

(F5). In this case, $\varphi = \varphi_1 \vee \varphi_2$, for some φ_1 and φ_2. Then, we know that $\Delta' = \Delta$, $\overline{c}' = \overline{c}$, $\vec{\varphi}' = \vec{\varphi}$ and either $\varphi' = \varphi_1$ or $\varphi' = \varphi_2$. In either of these cases we have $\mathcal{D} \models \forall\Delta.\, \varphi_j \Longrightarrow \varphi_1 \vee \varphi_2$, where $j \in \{1,2\}$. Thus $(\llbracket\mathcal{D}\rrbracket, V) \models \overline{c} \mathbin{\&} \vec{\varphi} \mathbin{\&} \varphi_j$ implies $(\llbracket\mathcal{D}\rrbracket, V) \models \overline{c} \mathbin{\&} \vec{\varphi} \mathbin{\&} (\varphi_1 \vee \varphi_2)$, for any $V \in \alpha\text{-}Tree_\Sigma(\Delta)$, as required.

(F6). We have that $\varphi = \exists x\!:\!E.\, \varphi^*$, for some φ^*, and by α-conversion we may assume that $x \notin dom(\Delta)$. Then, by rule (F6) we get that $\Delta' = \Delta, x\!:\!E$, $\overline{c}' = \overline{c}$, $\vec{\varphi}' = \varphi$ and $\varphi' = \varphi^*$. Given an arbitrary valuation $V \in \alpha\text{-}Tree_\Sigma(\Delta, x\!:\!E)$ we assume that $(\llbracket\mathcal{D}\rrbracket, V) \models \varphi^*$ holds. Writing V' for the restriction of V to $dom(\Delta)$, we get that $(\llbracket\mathcal{D}\rrbracket, V') \models \exists x\!:\!E.\, \varphi^*$, and then by Lemma 4.3.4 we have $(\llbracket\mathcal{D}\rrbracket, V) \models \exists x\!:\!E.\, \varphi^*$. Thus $\mathcal{D} \models \forall\Delta, x\!:\!E.\, (\overline{c} \mathbin{\&} \vec{\varphi} \mathbin{\&} \varphi^*) \Longrightarrow (\overline{c} \mathbin{\&} \vec{\varphi} \mathbin{\&} \exists x\!:\!E.\, \overline{c}^*)$, as required.

This completes the proof of Lemma 4.3.6. ☐

We are now in a position to prove the *logical soundness* half of Theorem 4.3.2.

Proof (of logical soundness). Let $\mathcal{D}$ be an α-inductive definition as in Definition 4.3.1. Given $(\Delta, \overline{c}, \vec{\varphi}, \varphi)$ such that $\Delta \vdash c\ ok$ for all $c \in \overline{c}$ and $\Delta \vdash \psi\ ok$ for all $\psi \in \vec{\varphi}, \varphi$, and given an arbitrary α-tree valuation $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ we assume that

$$\exists\Delta'(\overline{c}') \in solns_{\mathcal{D}}(\Delta, \overline{c}, \vec{\varphi}, \varphi) \tag{4.16}$$

$$(\llbracket\mathcal{D}\rrbracket, V) \models \exists\Delta'(\overline{c}') \tag{4.17}$$

both hold. By (4.16) and the definition of solution sets from Definition 4.3.1 there exist Δ' and $\overline{c}'$ such that

$$\mathcal{D} \vdash \exists\Delta(\overline{c}; \vec{\varphi}; \varphi) \rightsquigarrow \cdots \rightsquigarrow \exists\Delta, \Delta'(\overline{c}'; \emptyset; \top) \tag{4.18}$$

$$\models \exists\Delta, \Delta'(\overline{c}') \tag{4.19}$$

both hold. Then, by applying Lemma 4.3.6 to every individual $\rightsquigarrow$-transition in the sequence (4.18) we get that

$$\mathcal{D} \models \forall\Delta, \Delta'.\, \overline{c}' \Longrightarrow (\overline{c} \mathbin{\&} \vec{\varphi} \mathbin{\&} \varphi) \tag{4.20}$$

holds. Finally, by combining (4.17) and (4.20) we get $(\llbracket\mathcal{D}\rrbracket, V) \models (\overline{c} \mathbin{\&} \vec{\varphi} \mathbin{\&} \varphi)$, as required. ☐

4.3.2 Proof of logical completeness

In this section we prove the other half of Theorem 4.3.2, which we refer to as *logical completeness*. This result states that if a configuration $\exists\Delta(\overline{c}; \vec{\varphi}; \varphi)$ is satisfied by a valuation V then there exists a computation path in the αML operational semantics which terminates at an element $\exists\Delta'(\overline{c}')$ of the solution set which is satisfied by this valuation; hence αML does not discard any satisfying valuations during execution.

We first present a size metric on certain collections of satisfaction judgements, which we will use to show that formula reduction of satisfiable formulae eventually terminates. We begin by defining a size function $size(\varphi)$ on atomic formulae, as follows.

$$\begin{array}{rcl} size(r\,p) = size(\mathsf{T}) & = & 1 \\ size(c) & = & 2 \\ size(\varphi_1 \,\&\, \varphi_2) = size(\varphi_1 \vee \varphi_2) & = & 1 + size(\varphi_1) + size(\varphi_2) \\ size(\exists x\!:\!E.\,\varphi) & = & 1 + size(\varphi) \end{array}$$

It follows that $size(\varphi) \geq 1$ for all φ.

Definition 4.3.7 (Measure on satisfaction judgements). Recalling the definition of $[\![\mathcal{D}]\!]^{(n)}$ from Lemma 2.4.8 we write $\vec{\mathrm{J}}$ for a finite list of satisfaction judgements of the form $([\![\mathcal{D}]\!]^{(n_i)}, V) \models \varphi_i$, where the inductive definition $\mathcal{D}$ and the valuation V are the same in each judgement. For each natural number n we define $size_{\vec{\mathrm{J}}}(n)$ as follows.

$$size_{\vec{\mathrm{J}}}(n) \quad \triangleq \quad \sum_{(([\![\mathcal{D}]\!]^{(n)}, V) \models \varphi) \in \vec{\mathrm{J}}} size(\varphi)$$

Now, we write $\mu(\vec{\mathrm{J}})$ for the *multiset* of natural numbers which includes n with multiplicity $size_{\vec{\mathrm{J}}}(n)$, for every $n \in \mathbb{N}$. ◇

Intuitively, $size_{\vec{\mathrm{J}}}(n)$ records the total size of all formulae for which a satisfaction judgement using $[\![\mathcal{D}]\!]^{(n)}$ exists in $\vec{\mathrm{J}}$. The measure $\mu(\vec{\mathrm{J}})$ records a mapping from numbers n to these total sizes. Note that since $\vec{\mathrm{J}}$ is finite it follows that there are only finitely many n with non-zero multiplicity in $\mu(\vec{\mathrm{J}})$. We will use the multiset ordering construction of (Dershowitz and Manna, 1979) to derive a well-founded ordering on $\mu(\vec{\mathrm{J}})$. The following quote from that paper explains the intuition behind the multiset ordering.

> *The ordering $<$ on any given well-founded set S can be extended to form a well-founded ordering $\ll$ on the finite multisets over S. In this ordering, $M' \ll M$, for two finite multisets M and M' over S, if M' can be obtained from M by replacing one or more elements in M by any finite number of elements taken from S, each of which is smaller than one of the replaced elements. In particular, a multiset is reduced by replacing an element with zero elements, i.e. by deleting it.*

As a concrete example, it is the case that $\{1,1,1,2,2,2,3\} \ll \{1,2,3,3\}$ because one occurrence of 3 has been replaced by two occurrences of 2 and two occurrences of 1.

Therefore, we can derive a well-founded ordering $\prec$ on $\mu(\vec{\mathrm{J}})$, in terms of the $<$ ordering on the natural numbers. We will use this well-founded ordering to show that formula reduction of satisfiable formulae eventually terminates.

We now proceed to the main intermediate result in our proof of logical completeness, where we show that the set of formula reduction steps possible from a given configuration accounts for all satisfying valuations of that configuration. Moreover, we show that our well-founded measure on the list of satisfaction judgements strictly decreases across the formula reduction step.

Lemma 4.3.8. *Let $\mathcal{D} = \texttt{fun}\, r\texttt{(}x\texttt{:}S_r\texttt{)}\texttt{:prop =}\psi$ be an α-inductive definition as in Definition 4.3.1 and suppose that $\Delta \vdash \bar{c}\ \&\ \vec{\varphi}\ \&\ \varphi\texttt{:prop}$ and $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ both hold. Suppose that $\vec{\varphi} = \varphi_1\ \&\ \cdots\ \&\ \varphi_k$ and that $V \models \bar{c}$, $\forall i \in \{1,\ldots,k\}.\ (\llbracket\mathcal{D}\rrbracket^{(n_i)}, V) \models \varphi_i$ and $(\llbracket\mathcal{D}\rrbracket^{(n)}, V) \models \varphi$ all hold, for some $n_1,\ldots,n_k,n$. Then, either*

1. *$\vec{\varphi} = \emptyset$ and $\varphi = \mathsf{T}$; or*

2. *there exist Δ', $\bar{c}'$, $\vec{\varphi}'$, φ', V', $n'_1,\ldots,n'_j$ and n' such that*

$$\mathcal{D} \vdash \exists\Delta(\bar{c};\vec{\varphi};\varphi) \rightsquigarrow \exists\Delta,\Delta'(\bar{c}';\vec{\varphi}';\varphi') \tag{4.21}$$

$$V' \models \bar{c}' \tag{4.22}$$

$$\forall i \in \{1,\ldots,j\}.\ (\llbracket\mathcal{D}\rrbracket^{(n'_i)}, V) \models \varphi'_i \tag{4.23}$$

$$(\llbracket\mathcal{D}\rrbracket^{(n')}, V') \models \varphi' \tag{4.24}$$

all hold, where $\vec{\varphi}' = \varphi'_1\ \&\ \cdots\ \&\ \varphi'_j$ and where V' is an extension of V to $dom(\Delta,\Delta')$. Furthermore, if we write $\vec{\mathsf{J}}$ for $(\llbracket\mathcal{D}\rrbracket^{(n_1)}, V) \models \varphi_1,\ldots,(\llbracket\mathcal{D}\rrbracket^{(n_k)}, V) \models \varphi_k, (\llbracket\mathcal{D}\rrbracket^{(n)}, V) \models \varphi$ and $\vec{\mathsf{J}}'$ for $(\llbracket\mathcal{D}\rrbracket^{(n'_1)}, V') \models \varphi'_1,\ldots,(\llbracket\mathcal{D}\rrbracket^{(n'_j)}, V') \models \varphi'_j, (\llbracket\mathcal{D}\rrbracket^{(n')}, V') \models \varphi'$ then $\mu(\vec{\mathsf{J}}') \prec \mu(\vec{\mathsf{J}})$ holds also.

Proof. The proof is by case analysis on φ—the cases are as follows. We note that the case where $\varphi = \mathsf{F}$ cannot arise since $(R,V) \models \mathsf{F}$ is not derivable for any R, V.

Case $\varphi = r\,p$. In this case we use formula reduction rule (F1) to deduce that (4.21) holds where $\Delta' = \emptyset$, $\bar{c}' = \bar{c}$, $\vec{\varphi}' = \vec{\varphi}$ (i.e. $j = k$) and $\varphi' = \psi[p/x]$. If we set $V' = V$ and $n'_1 = n_1,\ldots,n'_j = n_j$ it is easy to see that (4.22) and (4.23) both hold.

Since $(\llbracket\mathcal{D}\rrbracket^{(n)}, V) \models r\,p$ it follows that $n = n'+1$ for some n' (since $\llbracket\mathcal{D}\rrbracket^{(0)} = \emptyset$ and $(\emptyset, V) \models r\,p$ is not derivable). Then, from $(\llbracket\mathcal{D}\rrbracket^{(n'+1)}, V) \models r\,p$ we get that $\llbracket p\rrbracket_V \in \llbracket\mathcal{D}\rrbracket^{(n'+1)}$. Using (2.6) and the definition of $\llbracket\mathcal{D}\rrbracket^{(n'+1)}$ we get that $(\llbracket\mathcal{D}\rrbracket^{(n')}, \{x \mapsto \llbracket p\rrbracket_V\}) \models \psi$. From Lemma 4.3.4(3) we get $(\llbracket\mathcal{D}\rrbracket^{(n')}, V[x \mapsto \llbracket p\rrbracket_V]) \models \psi$, and by Lemma 4.3.5 we have $(\llbracket\mathcal{D}\rrbracket^{(n')}, V) \models \psi[p/x]$, i.e. that (4.24) holds.

Finally, note that the multiset $\mu(\vec{\mathsf{J}}')$ is obtained from the multiset $\mu(\vec{\mathsf{J}})$ by replacing $size(r\,p) = 1$ occurrence of $n = n'+1$ with $size(\psi[p/x])$ occurrences of n'; hence $\mu(\vec{\mathsf{J}}') \prec \mu(\vec{\mathsf{J}})$.

Case $\varphi = c$. In this case our assumption tells us that $V \models \bar{c}\ \&\ c$, i.e. $\models \exists\Delta(\bar{c}\ \&\ c)$. Then, by (F2) we get that (4.21) holds, where $\Delta' = \emptyset$, $\bar{c}' = \bar{c}\ \&\ c$, $\vec{\varphi}' = \vec{\varphi}$ (i.e. $j = k$) and $\varphi' = \mathsf{T}$. If we set $V' = V$, $n' = n$, $n'_1 = n_1,\ldots,n'_j = n_j$ it follows that (4.22) and (4.23) both hold. We get that $\mu(\vec{\mathsf{J}}') \prec \mu(\vec{\mathsf{J}})$ holds because the multiplicity of n decreases by one, since $size(\mathsf{T}) < size(c)$.

Case $\varphi = \mathsf{T}$. In this case we perform a case split on the goal list $\vec{\varphi}$. If $\vec{\varphi} = \emptyset$ then we are immediately done, so we consider the case where $\vec{\varphi} = \vec{\varphi}'', \varphi''$ for some $\vec{\varphi}''$ and φ'', i.e. $k = k'+1$ for some k'. In this case, by (F3) we get that (4.21) holds, where $\Delta' = \emptyset$, $\bar{c}' = \bar{c}$, $\vec{\varphi}' = \vec{\varphi}''$ (i.e. $j = k'$) and $\varphi' = \varphi''$. Then, if we set $V' = V$, $n' = n_k$ and $n'_1 = n_1,\ldots,n'_j = n_j$ we get that (4.22) and (4.23) both hold, by assumption. Finally, since the T formula has been eliminated completely it follows that $\mu(\vec{\mathsf{J}}') \prec \mu(\vec{\mathsf{J}})$ holds, as required.

Case $\varphi = \varphi_1\ \&\ \varphi_2$. In this case we get that $(\llbracket\mathcal{D}\rrbracket^{(n)}, V) \models \varphi_m$ holds, for all $m \in \{1,2\}$. Then, by rule (F4) we get that (4.21) holds, where $\Delta' = \emptyset$, $\bar{c}' = \bar{c}$, $\vec{\varphi}' = \vec{\varphi}, \varphi_2$ (i.e. $j = k+1$) and $\varphi' = \varphi_1$. Then (4.22) and (4.23) both hold if we set $V' = V$, $n' = n$, $n'_1 = n_1,\ldots,n'_k = n_k$ and $n'_j = n$. Since $size(\varphi_1) + size(\varphi_2) < size(\varphi_1\ \&\ \varphi_2)$ it follows that $\mu(\vec{\mathsf{J}}') \prec \mu(\vec{\mathsf{J}})$ holds, as required.

Case $\varphi = \varphi_1 \vee \varphi_2$. Here we get that $([\![\mathcal{D}]\!]^{(n)}, V) \models \varphi_m$ holds, for some $m \in \{1,2\}$. Then, by rule (F5) we get that (4.21) holds when $\Delta' = \emptyset$, $\overline{c}' = \overline{c}$, $\vec{\varphi}' = \vec{\varphi}$ (i.e. $j = k$) and $\varphi' = \varphi_m$. If we set $V' = V$, $n' = n$ and $n'_1 = n_1, \dots, n'_k = n_k$ then (4.22) and (4.23) both hold. Finally, since $size(\varphi_j) < size(\varphi_1 \vee \varphi_2)$ we get that $\mu(\vec{\mathrm{J}}') \prec \mu(\vec{\mathrm{J}})$ holds, as required.

Case $\varphi = \exists x\!:\!E.\varphi''$. By α-renaming the bound variable in the formula we can assume that $x \notin dom(\Delta)$. Then we get that $([\![\mathcal{D}]\!]^{(n)}, V[x \mapsto t]) \models \varphi''$ holds, for some $t \in \alpha\text{-}Tree_\Sigma(E)$. By rule (F6) we get that (4.21) holds, if we let $\Delta = \{x\!:\!E\}$, $\overline{c}' = \overline{c}$, $\vec{\varphi}' = \vec{\varphi}$ (i.e. $j = k$) and $\varphi' = \varphi''$. If we also let $V' = V[x \mapsto t]$, $n' = n$ and $n'_1 = n_1, \dots, n'_k = n_k$ we can use Lemma 4.3.4(3) to show that (4.22) and (4.23) both hold. Also it follows that $\mu(\vec{\mathrm{J}}') \prec \mu(\vec{\mathrm{J}})$ holds, since $size(\varphi'') < size(\exists x\!:\!E.\,\varphi'')$.

This completes the proof of Lemma 4.3.8. □

Our proof of logical completeness rests on the fact that if $([\![\mathcal{D}]\!], V) \models \varphi$ holds then we can unfold the inductive definition, $\mathcal{D}$, n times (for some n) to produce an α-tree relation $[\![\mathcal{D}]\!]^{(n)}$ such that $([\![\mathcal{D}]\!]^{(n)}, V) \models \varphi$ holds.

Lemma 4.3.9. *Let $\mathcal{D} = \mathtt{fun}\, r(x\!:\!S_r)\!:\!\mathtt{prop} = \psi$ be an α-inductive definition as in Definition 4.3.1 and suppose that $\Delta \vdash \varphi\!:\!\mathtt{prop}$ and $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ both hold. Then $([\![\mathcal{D}]\!], V) \models \varphi$ holds iff $([\![\mathcal{D}]\!]^{(n)}, V) \models \varphi$ holds for some n.*

Proof. We proceed by induction on the structure of φ, using the fact that $[\![\mathcal{D}]\!] = \bigcup_{n \in \mathbb{N}} [\![\mathcal{D}]\!]^{(n)}$ (by Lemma 2.4.8). In the case of an atomic formula $r\,p$, we observe that if $(\mathcal{D}, V) \models r\,p$ then there exists n such that $\psi[p/x] \in [\![\mathcal{D}]\!]^{(n)}$, i.e. such that $([\![\mathcal{D}]\!]^{(n)}, V) \models r\,p$. In the case of a conjunction $\varphi_1 \,\&\, \varphi_2$ we use the fact that $[\![\mathcal{D}]\!]^{(n)} \subseteq [\![\mathcal{D}]\!]^{(n+1)}$ (which follows from the definition of $\Phi_{\mathcal{D}}$) to obtain a value n which is high enough to satisfy both φ_1 and φ_2. The other cases are straightforward. □

We now use Lemma 4.3.8 and Definition 4.3.1, along with Lemma 4.3.9, to present a proof of *logical completeness*.

Proof (of logical completeness). With the same assumptions as Definition 4.3.1, given an α-tree valuation $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ we assume that $([\![\mathcal{D}]\!], V) \models \overline{c} \,\&\, \vec{\varphi} \,\&\, \varphi$. It follows from Lemma 4.3.9 that we can find a list of satisfaction judgements $\vec{\mathrm{J}}$ as in the hypothesis of Lemma 4.3.8. Then, by Lemma 4.3.8 we can build up a sequence of formula reductions

$$\mathcal{D} \vdash \exists\Delta(\overline{c}; \vec{\varphi}; \varphi) \rightsquigarrow \cdots$$

with associated lists of satisfaction judgements $\vec{\mathrm{J}}$ of the form described in Definition 4.3.7. From Lemma 4.3.8 we know that at each step $\mu(\vec{\mathrm{J}})$ goes down in the well-founded ordering, therefore the sequence cannot be infinite and must eventually terminate. By Lemma 4.3.8 we get that the finite reduction sequence must be of the form

$$\mathcal{D} \vdash \exists\Delta(\overline{c}; \vec{\varphi}; \varphi) \rightsquigarrow \cdots \rightsquigarrow \exists\Delta, \Delta'(\overline{c}'; \emptyset; \mathsf{T})$$

where $V' \models \overline{c}'$ for some V' which extends V to $dom(\Delta, \Delta')$. Thus, by Definition 4.3.1, we have shown that there is some $\exists\Delta'(\overline{c}') \in solns_{\mathcal{D}}(\Delta, \overline{c}, \vec{\varphi}, \varphi)$ such that $([\![\mathcal{D}]\!], V) \models \exists\Delta'(\overline{c}')$ holds, as required. □

This result, along with the *logical soundness* result proved in Section 4.3.1 above, gives us Theorem 4.3.2. Therefore the operational semantics of αML computes all and only the solutions to an initial "query" formula ($\overline{c} \,\&\, \vec{\varphi} \,\&\, \varphi$), expressed as an αML formula reduction configuration $\exists\Delta(\overline{c}; \vec{\varphi}; \varphi)$. The relative simplicity of these proofs is a demonstration of the power of

Theorem 4.2.2, because when doing proofs about embedded formulae we can forget about the details of the αML operational semantics which are not relevant and just focus on the subset of formula reduction transitions.

The *algebraic* soundness and completeness results described in (Jaffar et al., 1998) are weaker than the results that we have already proved. Those results express that termination in the operational semantics corresponds to satisfiability in the logical semantics of formulae, but do not relate the set of satisfying valuations to the results of the computations in the meta-language. The corresponding algebraic soundness and completeness results for αML are stated below, and and follow directly from Theorem 4.2.2 and Theorem 4.3.2.

Theorem 4.3.10 (Algebraic soundness and completeness). *With $\mathcal{D}$ and $(\Delta, \overline{c}, \vec{\varphi}, \vec{\varphi})$ defined as in as in Definition 4.3.1, it is the case that*

$$\models \exists\Delta(\overline{c} \;\&\; \vec{\varphi} \;\&\; \varphi) \iff \exists\Delta(\overline{c}; F_{\vec{\varphi}'}[v_{\mathcal{D}}/r]; \varphi[v_{\mathcal{D}}/r])\downarrow.$$

The forward direction is algebraic completeness *and the reverse is* algebraic soundness. □

4.4 Example definition

We conclude this chapter with an example of an α-inductive definition encoded as a recursive function in αML. We recall the `ftv` relation defined in Example 2.4.3, which encodes the notion of "free type variables" in a System F type. The following αML code is the recursive function which corresponds to the $\mathcal{D}_{\text{ftv}}$ inductive definition.

```
fun r(x:S_r):prop = (∃a:tyvar.∃a':tyvar. x = ftv (a, TyVar a') & a # a')
  ∥ (∃a:tyvar.∃τ1:type.∃τ2:type.
      x = ftv (a, Fun (τ1, τ2)) & r (ftv (a, τ1)) & r (ftv (a, τ2)))
  ∥ (∃a:tyvar.∃τ:type. x = ftv (a, ForAll<a>τ) & ⊤)
  ∥ (∃a:tyvar.∃a':tyvar.∃τ:type. x = ftv (a, ForAll<a'>τ) & a # a' & r (ftv (a, τ)))
```

The structure of this function closely mirrors that of the schematic rule (in standard form) from Example 2.4.3. This highlights the close relationship between αML syntax and the syntax of α-inductive definitions. When the function is called, a non-deterministic branch is spawned for each inference rule and the constraint solving procedure determines whether a given rule can be matched against. If so, the code corresponding to the premises is executed, which may include some recursive calls to the function r.

The code size is already starting to get large for even this simple example, so we will not present αML code for any larger α-inductive definitions. In the αML implementation this elaboration is carried out automatically so users can write larger definitions in rule-like syntax: see Appendix D for a larger example.

Chapter 5

Contextual equivalence

"All animals are equal but some animals are more equal than others."
—G. Orwell

In this chapter we develop a notion of contextual equivalence for αML expressions. We define an operational equivalence relation which holds between two expressions when they behave identically in all configurations. We take identical behaviour to mean that either both configurations terminate with success or neither configuration terminates with success. We prove that the operational equivalence relation has certain desirable properties which imply that it coincides with contextual equivalence.

We then demonstrate an encoding of ground trees into αML and prove that two ground trees are α-equivalent precisely when their encodings are operationally equivalent. This is a "correctness of representation" result similar to those proved for FreshML (Shinwell and Pitts, 2005; Pitts and Shinwell, 2008). We also relate semantic equivalence of schematic formulae to operational equivalence of the corresponding αML expressions and discuss an alternative equivalence relation which also observes finite failure. We close the chapter with a brief discussion of fresh name generation.

5.1 Definition of operational equivalence

In this section we define a notion of operational equivalence for αML expressions.

Definition 5.1.1 (Operational equivalence). Recalling the definition of success from Definition 3.7.1, we define the operational equivalence relation $\boldsymbol{\Delta \vdash e \cong e' : T}$ which holds iff

- $\Delta \vdash e : T$ and $\Delta \vdash e' : T$; and
- $\exists \boldsymbol{\Delta'}(\overline{c}; F; e)\downarrow \iff \exists \boldsymbol{\Delta'}(\overline{c}; F; e')\downarrow$ holds for all Δ', $\overline{c}$, F and T' such that $\Delta' \supseteq \Delta$ and $\Delta' \vdash \overline{c} : \texttt{prop}$ and $\Delta' \vdash F : T \to T'$. $\diamond$

We extend this definition to a relation $\cong^{\circ}$ between arbitrary αML expressions, including those which contain free variables that are not of equality types. We will refer to $\cong^{\circ}$ as the *open extension* of $\cong$, even though both relations contain expressions with free variables. The open extension is defined in terms of $\cong$ by substituting values which are "closed" (in the sense that the only contain free variables of equality types) for the free variables which are not of an equality type.

Definition 5.1.2 (Open extension of $\cong$). Let the typing environment Γ be decomposed into disjoint typing environments Δ and Γ', where $\Gamma'(x)$ is not an equality type for any $x \in dom(\Gamma')$. Then, the open extension of operational equivalence $\boldsymbol{\Gamma \vdash e \cong^\circ e' : T}$ holds iff $\boldsymbol{\Delta' \vdash e[\sigma] \cong e'[\sigma] : T}$ holds for all $\Delta' \supseteq \Delta$ and all $\sigma \in Sub_\Sigma(\Gamma', \Delta')$. ◇

Note that the $\cong^\circ$ relation only observes success (Definition 3.7.1), not finite failure (Definition 3.7.2). In Section 5.6 we will present a finer-grained notion of operational equivalence (which we will denote by $\cong^\circ_F$) that allows both success and finite failure to be observed.

Remark 5.1.3 (Operational equivalence of pure expressions). In Section 3.6 we showed that the pure transition relation $\rightarrow_P$ carves out a subset of αML transitions which correspond to the evaluation of a standard, strict functional programming language. It follows that a restricted operational equivalence relation exists within the sub-class of pure expressions (Definition 3.6.1) which coincides with the standard, well-studied notion of contextual equivalence for this language. ◇

5.2 Expression relations

Before we consider the properties of the operational equivalence relation defined above, we first present a general notion of type-respecting relations between αML expressions. This technique has been used in (Pitts, 2005) and (Pitts and Shinwell, 2008).

Definition 5.2.1 (Expression relations). An *expression relation* $\mathcal{E}$ is a set of tuples (Γ, e, e', T), made up of a typing environment, two expressions and a type, such that $\Gamma \vdash e : T$ and $\Gamma \vdash e' : T$. We write $\Gamma \vdash e\,\mathcal{E}\,e' : T$ to mean that $(\Gamma, e, e', T) \in \mathcal{E}$. We now enumerate some standard properties of expression relations. We say that

- $\mathcal{E}$ is an **equivalence relation** if it is *reflexive* ($\Gamma \vdash e : T \implies \Gamma \vdash e\,\mathcal{E}\,e : T$), *symmetric* ($\Gamma \vdash e\,\mathcal{E}\,e' : T \implies \Gamma \vdash e'\,\mathcal{E}\,e : T$) and *transitive* ($\Gamma \vdash e\,\mathcal{E}\,e' : T \wedge \Gamma \vdash e'\,\mathcal{E}\,e'' : T \implies \Gamma \vdash e\,\mathcal{E}\,e'' : T$).
- $\mathcal{E}$ has the **weakening** property if $\Gamma \vdash e\,\mathcal{E}\,e' : T$ and $\Gamma' \supseteq \Gamma$ imply $\Gamma' \vdash e\,\mathcal{E}\,e' : T$.
- $\mathcal{E}$ is **substitutive** if $\Gamma, \Gamma' \vdash e\,\mathcal{E}\,e' : T$ and $\Gamma \vdash \sigma\,\mathcal{E}\,\sigma' : \Gamma'$ imply $\Gamma \vdash e[\sigma]\,\mathcal{E}\,e'[\sigma] : T$, where $\Gamma \vdash \sigma\,\mathcal{E}\,\sigma' : \Gamma'$ means that $\sigma, \sigma' \in Sub_\Sigma(\Gamma, \Gamma')$ and that $\Gamma' \vdash \sigma(x)\,\mathcal{E}\,\sigma'(x) : \Gamma(x)$ holds for all $x \in dom(\Gamma)$.
- $\mathcal{E}$ is **compatible** if $\widehat{\mathcal{E}} \subseteq \mathcal{E}$, where $\widehat{\mathcal{E}}$ is the *compatible refinement* of $\mathcal{E}$. This operation on expression relations is defined in Figure 5.1.
- $\mathcal{E}$ is **adequate** if $\Delta \vdash e\,\mathcal{E}\,e' : T$ implies $\Delta \vdash e \cong e' : T$.

Most of these definitions are standard. The most interesting is compatibility, which states that membership of the expression relation is preserved by the term-formers of the αML language. We note in particular that operational equivalence ($\cong^\circ$) is an expression relation because it requires that the two expressions both have the same type. A property of expression relations that is stated explicitly in (Pitts and Shinwell, 2008) but omitted from Definition 5.2.1 is equivariance. It is trivial to show that all αML expression relations are equivariant (in the sense of Section 2.5) because names do not appear in the syntax of αML.

We extend the definition of compatible refinement to frame stacks and constraint problems, as shown in Figure 5.2. If two frame stacks are related by $\widehat{\mathcal{E}}$ then they have the same length and contain contain $\mathcal{E}$-related expressions at corresponding points in the stack. Similarly, two $\widehat{\mathcal{E}}$-related constraint problems contain the same number of constraints, which are $\mathcal{E}$-related in a similar way.

$$\frac{x \in dom(\Gamma) \qquad \Gamma(x) = T}{\Gamma \vdash x \mathrel{\widehat{\mathcal{E}}} x : T} \qquad\qquad \frac{(K : T \to D) \in \Sigma \qquad \Gamma \vdash v \mathrel{\mathcal{E}} v' : T}{\Gamma \vdash K\, v \mathrel{\widehat{\mathcal{E}}} K\, v' : D}$$

$$\frac{\Gamma \vdash v_1 \mathrel{\mathcal{E}} v_1' : T_1 \qquad \cdots \qquad \Gamma \vdash v_n \mathrel{\mathcal{E}} v_n' : T_n}{\Gamma \vdash (v_1, \ldots, v_n) \mathrel{\widehat{\mathcal{E}}} (v_1', \ldots, v_n') : T_1 * \cdots * T_n} \qquad\qquad \frac{}{\Gamma \vdash \texttt{()} \mathrel{\mathcal{E}} \texttt{()} : \texttt{unit}}$$

$$\frac{\Gamma, f : T \to T', x : T \vdash e \mathrel{\mathcal{E}} e' : T' \qquad f, x \notin dom(\Gamma)}{\Gamma \vdash (\texttt{fun}\, f\texttt{(}x : T\texttt{)} : T' \texttt{=} e) \mathrel{\widehat{\mathcal{E}}} (\texttt{fun}\, f\texttt{(}x : T\texttt{)} : T' \texttt{=} e') : T \to T'} \qquad\qquad \frac{}{\Gamma \vdash \top \mathrel{\widehat{\mathcal{E}}} \top : \texttt{prop}}$$

$$\frac{\Gamma \vdash v_1 \mathrel{\mathcal{E}} v_1' : N \qquad \Gamma \vdash v_2 \mathrel{\mathcal{E}} v_2' : E}{\Gamma \vdash \texttt{<}v_1\texttt{>}v_2 \mathrel{\widehat{\mathcal{E}}} \texttt{<}v_1'\texttt{>}v_2' : \texttt{[}N\texttt{]}\, E} \qquad\qquad \frac{\Gamma \vdash v_1 \mathrel{\mathcal{E}} v_1' : E \qquad \Gamma \vdash v_2 \mathrel{\mathcal{E}} v_2' : E}{\Gamma \vdash (v_1 \texttt{=} v_2) \mathrel{\widehat{\mathcal{E}}} (v_1' \texttt{=} v_2') : \texttt{prop}}$$

$$\frac{\Gamma \vdash v_1 \mathrel{\mathcal{E}} v_1' : N \qquad \Gamma \vdash v_2 \mathrel{\mathcal{E}} v_2' : E}{\Gamma \vdash (v_1 \,\texttt{\#}\, v_2) \mathrel{\widehat{\mathcal{E}}} (v_1' \,\texttt{\#}\, v_2') : \texttt{prop}} \qquad\qquad \frac{\Gamma \vdash e_1 \mathrel{\mathcal{E}} e_1' : T \qquad \Gamma, x : T \vdash e_2 \mathrel{\mathcal{E}} e_2' : T' \qquad x \notin dom(\Gamma)}{\Gamma \vdash (\texttt{let}\, x \texttt{=} e_1 \,\texttt{in}\, e_2) \mathrel{\widehat{\mathcal{E}}} (\texttt{let}\, x \texttt{=} e_1' \,\texttt{in}\, e_2') : T'}$$

$$\frac{\Gamma \vdash v_1 \mathrel{\mathcal{E}} v_1' : T \to T' \qquad \Gamma \vdash v_2 \mathrel{\mathcal{E}} v_2' : T}{\Gamma \vdash (v_1\, v_2) \mathrel{\widehat{\mathcal{E}}} (v_1'\, v_2') : T'}$$

$$\frac{\begin{array}{c} x_1 \neq \ldots \neq x_n \notin dom(\Gamma) \qquad D = K_1 \,\texttt{of}\, T_1 \mid \cdots \mid K_n \,\texttt{of}\, T_n \\ \Gamma \vdash v \mathrel{\mathcal{E}} v' : D \qquad \Gamma, x_1 : T_1 \vdash e_1 \mathrel{\mathcal{E}} e_1' : T \qquad \cdots \qquad \Gamma, x_n : T_n \vdash e_n \mathrel{\mathcal{E}} e_n' : T \end{array}}{\begin{array}{c} \Gamma \vdash (\texttt{case}\, v \,\texttt{of}\, K_1\, x_1 \texttt{->} e_1 \mid \cdots \mid K_n\, x_n \texttt{->} e_n) \mathrel{\widehat{\mathcal{E}}} \\ (\texttt{case}\, v' \,\texttt{of}\, K_1\, x_1 \texttt{->} e_1' \mid \cdots \mid K_n\, x_n \texttt{->} e_n') : T \end{array}}$$

$$\frac{\Gamma \vdash v \mathrel{\mathcal{E}} v' : T_1 * \cdots * T_n \qquad i \in \{1, \ldots, n\}}{\Gamma \vdash (v.i) \mathrel{\widehat{\mathcal{E}}} (v'.i) : T_i} \qquad\qquad \frac{\Gamma \vdash e_1 \mathrel{\mathcal{E}} e_1' : T \qquad \Gamma \vdash e_2 \mathrel{\mathcal{E}} e_2' : T}{\Gamma \vdash (e_1 \parallel e_2) \mathrel{\widehat{\mathcal{E}}} (e_1' \parallel e_2') : T}$$

$$\frac{x \notin dom(\Gamma) \qquad \Sigma \vdash E \; inhab \qquad \Gamma, x : E \vdash e \mathrel{\mathcal{E}} e' : T}{\Gamma \vdash (\exists x : E.\, e) \mathrel{\widehat{\mathcal{E}}} (\exists x : E.\, e') : T}$$

Figure 5.1: Compatible refinement $\widehat{\mathcal{E}}$ of an expression relation $\mathcal{E}$

$$\frac{}{\Gamma \vdash \texttt{Id} \mathrel{\widehat{\mathcal{E}}} \texttt{Id} : T \to T} \qquad\qquad \frac{\Gamma, x : T \vdash e \mathrel{\mathcal{E}} e' : T' \qquad \Gamma \vdash F \mathrel{\widehat{\mathcal{E}}} F' : T' \to T'' \qquad x \notin dom(\Gamma)}{\Gamma \vdash (F \circ (x.e)) \mathrel{\widehat{\mathcal{E}}} (F' \circ (x.e')) : T \to T''}$$

$$\frac{\begin{array}{c} \overline{c} = c_1 \,\texttt{\&}\, \cdots \,\texttt{\&}\, c_n \qquad \overline{c}' = c_1' \,\texttt{\&}\, \cdots \,\texttt{\&}\, c_n' \\ \Gamma \vdash c_1 \mathrel{\mathcal{E}} c_1' : \texttt{prop} \qquad \cdots \qquad \Gamma \vdash c_n \mathrel{\mathcal{E}} c_n' : \texttt{prop} \end{array}}{\Gamma \vdash \overline{c} \mathrel{\widehat{\mathcal{E}}} \overline{c}' : \texttt{prop}}$$

Figure 5.2: Extension of compatible refinement to frame stacks and constraint problems

The following lemma enumerates some general properties of the compatible refinement rules defined in Figure 5.1 and Figure 5.2, which is that the rules preserve the reflexivity, symmetry and weakening properties. The results are proved by long inductions over the compatible refinement rules from those figures.

Lemma 5.2.2. *The compatible refinement operator preserves the reflexivity, symmetry and weakening properties, i.e. if $\mathcal{E}$ is reflexive or symmetric, or has the weakening property then $\widehat{\mathcal{E}}$ has the corresponding property also.* □

5.3 CIU theorem

In this section we prove a "CIU" theorem for the operational equivalence relation $\cong^\circ$. This enumerates some desirable properties of our relation, following the technique of (Mason and Talcott, 1991). The mnemonic stands for "Uses of closed instantions", because Definition 5.1.2 quantifies over all possible closing substitutions before we test the termination behaviour of expressions.

Theorem 5.3.1 (CIU). *The operational equivalence relation $\cong^\circ$, is an equivalence relation and has the weakening property. Furthermore, it is adequate, substitutive and compatible. It is also the largest such expression relation.*

Proof. We demonstrate these properties of the $\cong^\circ$ relation individually.

- **$\cong^\circ$ is an equivalence relation.**
  It is clear that the $\cong$ relation is an equivalence relation, by breaking down the definition from Definition 5.1.1 (since $\iff$ is an equivalence relation).

- **$\cong^\circ$ has the weakening property.**
  This follows by expanding out the definition of $\cong^\circ$ in terms of $\cong$ and the termination of αML configurations.

- **$\cong^\circ$ is adequate.**
  Recall that an expression relation $\mathcal{E}$ is adequate if $\Delta \vdash e \,\mathcal{E}\, e' : T$ implies $\Delta \vdash e \cong e' : T$. This is clearly the case for $\cong^\circ$, by its definition in terms of $\cong$.

- **$\cong^\circ$ is substitutive**
  It suffices to prove the case where the typing environment Γ maps all variables in $dom(\Gamma)$ to equality types, and where the substitutions are both singletons, i.e. we aim to show that

  $$\Delta, x : T \vdash e \cong^\circ e' : T' \;\wedge\; \Delta \vdash v \cong v' : T \implies \Delta \vdash e[v/x] \cong e'[v'/x] : T'.$$

  This is sufficient because we can repeatedly apply this result to simulate any closing substitution, including those used to define $\cong^\circ$ in terms of $\cong$. We therefore assume that $\Delta, x : T \vdash e \cong^\circ e' : T'$ and $\Delta \vdash v \cong v' : T$ both hold. By choosing appropriate configurations we may infer that

  $$\exists\Delta'(\overline{c}; F; e[v'/x])\downarrow \iff \exists\Delta'(\overline{c}; F; e'[v'/x])\downarrow \quad (5.1)$$
  $$\exists\Delta'(\overline{c}; F \circ (x.\, e); v)\downarrow \iff \exists\Delta'(\overline{c}; F \circ (x.\, e); v')\downarrow \quad (5.2)$$

  both hold, where $\Delta' \supseteq \Delta$, $\Delta' \vdash \overline{c} : \texttt{prop}$ and $\Delta' \vdash F : T' \to T''$ all hold, for some Δ', $\overline{c}$, F and T''. Then, using rules (I1) and (P1) we can make a $\longrightarrow$ transition on both sides of (5.2) to get that $\exists\Delta'(\overline{c}; F; e[v/x])\downarrow \iff \exists\Delta'(\overline{c}; F; e[v'/x])\downarrow$ holds. Then, we can combine this with (5.1) to get $\exists\Delta'(\overline{c}; F; e[v/x])\downarrow \iff \exists\Delta'(\overline{c}; F; e'[v'/x])\downarrow$, and finally by Definition 5.1.2 we get that $\Delta \vdash e[v/x] \cong e'[v'/x] : T'$ holds, as required.

- **$\cong^\circ$ is compatible**
  This proof is rather long so it is presented in Appendix A.

- **$\cong^\circ$ is the largest expression relation with the above properties**
  We seek to prove that, for all expression relations $\mathcal{E}$ with the above properties, $\mathcal{E} \subseteq \cong^\circ$. We fix an expression relation $\mathcal{E}$, and assume that $\mathcal{E}$ is reflexive, symmetric and transitive, adequate, substitutive and compatible, and that $\mathcal{E}$ has the weakening property. Now, suppose that $\Gamma \vdash e \,\mathcal{E}\, e' : T$ holds, and that $\Gamma = \Delta, \Gamma'$, where $\Gamma'(x)$ is not an equality type for all $x \in dom(\Gamma')$. Now, let σ be an arbitrary substitution from the set $Sub_\Sigma(\Gamma', \Delta')$, for some $\Delta' \supseteq \Delta$. Since $\mathcal{E}$ is reflexive we know that $\Gamma' \vdash \sigma \,\mathcal{E}\, \sigma : \Delta'$ holds. Then, by the fact that $\mathcal{E}$ is substitutive and has the weakening property, we get that $\Delta' \vdash e[\sigma] \,\mathcal{E}\, e'[\sigma] : T$ holds, and since $\mathcal{E}$ is adequate we know that $\Delta' \vdash e[\sigma] \cong e'[\sigma] : T$. Finally, by definition of $\cong^\circ$, this is equivalent to $\Gamma \vdash e \cong^\circ e' : T$, as required.

This completes the proof of Theorem 5.3.1. □

Following the approach of (Gordon, 1998) and (Lassen, 1998), we have therefore shown that the $\cong^\circ$ relation coincides with the standard notion of contextual equivalence:

$$\Gamma \vdash e \cong_{\text{ctx}} e' : T \quad \triangleq \quad \Gamma \vdash e : T \wedge \Gamma \vdash e' : T \wedge (\forall \mathcal{C} \in Ctx_\Sigma(T).\, \mathcal{C}[e]\!\downarrow \iff \mathcal{C}[e']\!\downarrow)$$

where $Ctx_\Sigma(T)$ is the set of all αML program contexts $\mathcal{C}$ which accept values of type T (the definition is straightforward given the language syntax defined in Figure 3.1). The CIU theorem shows that $\cong^\circ$ possesses the key properties of contextual equivalence, being the largest congruence relation (compatible, substitutive equivalence relation) which contains $\cong^\circ$. Henceforth we will refer to $\cong^\circ$ as *contextual equivalence*.

Having established the basic properties of program equivalence, we can begin to derive some specific instances of useful equivalences. These could be used to implement provably correct compiler transformations on αML programs (although the αML implementation described in Chapter 7 does not perform any such optimisations). We enumerate a few such results below, without proof.

Lemma 5.3.2 (Order of variable generation). *If* $\Gamma \vdash \exists x_1 : E_1.\, \exists x_2 : E_2.\, e : T$ *then* $\Gamma \vdash (\exists x_1 : E_1.\, \exists x_2 : E_2.\, e) \cong^\circ (\exists x_2 : E_2.\, \exists x_1 : E_1.\, e) : T$. □

Lemma 5.3.3 (Order of constraint processing). *If* $\Gamma \vdash c_1 : \texttt{prop}$ *and* $\Gamma \vdash c_2 : \texttt{prop}$ *then* $\Gamma \vdash (c_1 \,\&\, c_2) \cong^\circ (c_2 \,\&\, c_1) : \texttt{prop}$. □

Lemma 5.3.4 (Scope extrusion). *If* $\Gamma \vdash \exists x : E.\, e : T$, $\Gamma \vdash e' : T'$ *and* $x \notin dom(\Gamma)$ *then* $\Gamma \vdash ((\exists x : E.\, e) \,\&\, e') \cong^\circ (\exists x : E.\, (e \,\&\, e')) : T'$. □

5.4 Correctness of data representation

We now show that two ground trees g and g' are α-equivalent precisely when their encodings $[\![g]\!]$ and $[\![g']\!]$ are contextually equivalent in αML. This is a fundamental correctness result if we claim to support meta-programming with binders handled correctly modulo α-equivalence. A similar result was proved for FreshML (Shinwell, 2005; Pitts and Shinwell, 2008) but the proof presented here is substantially different (and, arguably, simpler) because the language constructs of αML are different to those of FreshML. The data correctness result for αML is interesting as it shows that it is possible to faithfully represent abstract syntax modulo α-equivalence using only variables and locally-asserted freshness constraints, without the use of permutative names and the corresponding notion of global freshness.

5.4.1 Names and variables

We recall the definition of ground trees from Definition 2.2.1:

$$g \quad ::= \quad n \mid \texttt{<}n\texttt{>}g \mid () \mid (g_1, \ldots, g_n) \mid K\, g.$$

The only identifiers which appear in ground trees are names, which do not appear in the syntax of αML expressions. We therefore fix a bijection between the countably infinite sets of names (*Name*) and variables (*Var*). We write $\mathcal{V}(n)$ for the variable corresponding to the name n (we do not require the reverse direction for this exposition). The fixed bijection will be used to translate the free names of a ground tree. To deal with the bound names, we introduce a notion of name environments.

Definition 5.4.1 (Name environments). Let ε range over *name environments*, which are finite partial functions from the set of names (*Name*) to the set of variables (*Var*). We write $\varepsilon[n \mapsto x]$ for the environment which maps n to x but otherwise behaves like ε. In particular, we will write ε_g for the environment $\{n \mapsto \mathcal{V}(n) \mid n \in FN(g)\}$ which maps the free names of g to variables according to the fixed bijection. We write $dom(\varepsilon)$ and $cod(\varepsilon)$ for the domain and codomain of ε, respectively. ◇

The names in a ground tree are always of a name sort—hence the variables occurring in translated trees should also all be of name sort. We therefore let η range over type environments which only map variables to name sorts. (This is a strict subset of the type environments ranged over by Δ.)

We now pick out some important relationships between name environments and type environments. Throughout, we write $\overline{n}$ for a finite set of distinct names. For any name environment ε we write η_ε for the corresponding type environment $\{\varepsilon(n){:}sort(n) \mid n \in dom(\varepsilon)\}$. Furthermore, we will write $\Gamma \vdash_\varepsilon \overline{n}$ to mean that $\Gamma(\varepsilon(n)) = sort(n)$ for all $n \in \overline{n}$, i.e. the type environment Γ respects the name-sorting function on the image of $\overline{n}$ under ε. For the special case where ε is the (appropriate subset of the) fixed bijection $\mathcal{V}(n)$ between names and variables, we will elide the environment and just write $\Gamma \vdash \overline{n}$.

5.4.2 Representation of ground trees

The first step towards proving the data correctness theorem for αML is to define a representation of ground trees in αML. We begin by defining a relation $\varepsilon \vdash \langle \eta, g\rangle \triangleright \langle \eta', v\rangle$, where $FN(g) \subseteq dom(\varepsilon)$, $\eta \vdash_\varepsilon dom(\varepsilon)$, $\eta' \supseteq \eta$ and $\eta' \vdash v{:}E$ where $g \in Tree_\Sigma(E)$. The informal meaning of this judgement is that g can be translated into v, where the free names of g are represented using the mappings in ε. Each bound name in g is represented by a distinct variable in $dom(\eta') - dom(\eta)$ and η' assigns name sorts to all of the variables in η together with those which represent the bound names. The name environment ensures that the binding scope of the names from the tree is respected. The $\triangleright$ relation is defined by the rules in Figure 5.3.

The type environment is threaded through the definition as state. For example, in the rule for tuples the type environments η_i are passed along the subsequent premises of the rule. In contrast, each of the premises of this rule involves the same name environment ε. This reflects the fact that the scope of name-bindings is a structural property of the syntax of the tree. The name environment is only used by the rule for names, which looks up a name in the environment and translates it to the appropriate variable.

The abstraction rule modifies both the type and name environments. For an abstraction $\texttt{<}n\texttt{>}g$, the type environment is enlarged by choosing a fresh variable which does not already occur in η and adding it, with the same type as n (i.e. $sort(n)$). We note that the side-condition

$$\frac{\varepsilon(n) = x}{\varepsilon \vdash \langle \eta, n \rangle \rhd \langle \eta, x \rangle} \qquad \frac{}{\varepsilon \vdash \langle \eta, \texttt{()} \rangle \rhd \langle \eta, \texttt{()} \rangle} \qquad \frac{\varepsilon \vdash \langle \eta, g \rangle \rhd \langle \eta', v \rangle}{\varepsilon \vdash \langle \eta, K\, g \rangle \rhd \langle \eta', K\, v \rangle}$$

$$\frac{\varepsilon \vdash \langle \eta, g_1 \rangle \rhd \langle \eta_1, v_1 \rangle \quad \cdots \quad \varepsilon \vdash \langle \eta_{k-1}, g_k \rangle \rhd \langle \eta', v_k \rangle}{\varepsilon \vdash \langle \eta, \texttt{(}g_1 \texttt{,} \ldots \texttt{,} g_k\texttt{)} \rangle \rhd \langle \eta', \texttt{(}v_1 \texttt{,} \ldots \texttt{,} v_k\texttt{)} \rangle}$$

$$\frac{x \notin dom(\eta) \qquad \varepsilon[n \mapsto x] \vdash \langle \eta, x\!:\!sort(n), g \rangle \rhd \langle \eta', v \rangle}{\varepsilon \vdash \langle \eta, \texttt{<}n\texttt{>}g \rangle \rhd \langle \eta', \texttt{<}x\texttt{>}v \rangle}$$

Figure 5.3: Tree translation rules

in the abstraction rule $(x \notin dom(\eta))$ can always be satisfied because we can always find a new variable that is not in the finite domain of η. The abstraction rule modifies the name environment by overriding any existing mapping for n so that it is mapped to the freshly-generated variable x. This corresponds to the lexical scope of that bound occurrence of n in the original tree. There may well be multiple occurrences of n, even multiple bound occurrences: this is not problematic because each binding occurrence gets implemented by a distinct variable in αML and the environment ensures that the binding structure of the tree is faithfully represented.

The type environments η are not actually used anywhere in the $\rhd$ rules, but serve to record the variables which appear in the value that corresponds to g. Keeping track of the set of variables used locally to represent names allows us to manually assert freshnesses about particular names. The translation will make use of the notation from Definition 2.5.8 for expressing that a set of variables (of name sort) should be mutually distinct. This follows the intuition that "what matters about names when they are used to describe binding structure is not their particular identity, but rather the *distinctions* between them" (Lakin and Pitts, 2009). We can now use the relation from Figure 5.3 to define the translation of ground trees into αML.

Definition 5.4.2 (Tree translation). For a ground tree g, its αML translation $[\![g]\!]$ is defined as

$$[\![g]\!] \;\triangleq\; \exists(\eta' - \eta_{\varepsilon_g}).\, \#_{dom(\eta')} \;\&\; v_g$$

where $\varepsilon_g \vdash \langle \eta_{\varepsilon_g}, g \rangle \rhd \langle \eta', v_g \rangle$. ◇

The αML representation $[\![g]\!]$ of a ground tree g is not a value but rather an expression which, when evaluated, creates a pattern v_g that reflects the structure of g and generates constraints that the variables standing for distinct free and bound names of g must be pairwise distinct. The main theorems of this section will demonstrate that, taken together, the value and the constraints faithfully represent the structure and binding of the tree g. As a specific example, the αML encoding of the ground tree $\texttt{(<}n\texttt{><}n\texttt{>}n\texttt{, (}n\texttt{, }n'\texttt{))}$ is

$$\exists x_1\!:\!N.\, \exists x_2\!:\!N.\, \#_{\{x_1, x_2, \mathcal{V}(n), \mathcal{V}(n')\}} \;\&\; \texttt{(<}x_1\texttt{><}x_2\texttt{>}x_2\texttt{, (}\mathcal{V}(n)\texttt{, }\mathcal{V}(n')\texttt{))} \tag{5.3}$$

where we assume that $sort(x_1) = sort(x_2) = N$. This example illustrates the handling of multiple, nested occurrences of the same name in abstraction position and furthermore a clash between a bound and a free name.

Remark 5.4.3 (Possibility of failure). The translation of ground trees into αML imposes freshness constraints on the variables which appear in the αML encoding. The variables which

correspond to free names in a ground tree g are free variables of the corresponding αML expression $[\![g]\!]$, so the surrounding evaluation context could impose additional constraints on these variables. This means that the evaluation of a translated ground tree could fail finitely in αML. For example, evaluating the translated ground tree from (5.3) in the configuration

$$\exists \mathcal{V}(n):N, \mathcal{V}(n'):N(\mathcal{V}(n) = \mathcal{V}(n'); \texttt{Id}; -)$$

will fail because the freshness constraint between $\mathcal{V}(n)$ and $\mathcal{V}(n')$ in the translated expression is inconsistent with the existing constraints in the configuration. This issue does not affect the truth of the data correctness theorem for αML because if two ground trees g and g' are α-equivalent they have the same set of free names, so if $[\![g]\!]$ fails finitely when evaluated then $[\![g']\!]$ will too. $\diamond$

Lemma 5.4.4 (Well-formedness for tree translations).

- *If* $g \in Tree_\Sigma(E)$ *and* $\varepsilon \vdash \langle \eta, g \rangle \triangleright \langle \eta', v \rangle$ *then* $\eta' \vdash v : E$.
- *If* $g \in Tree_\Sigma(E)$ *and* $\Gamma \vdash FN(g)$ *then* $\Gamma \vdash [\![g]\!] : E$.

Proof. The first property can be proved by induction on the rules defining the $\triangleright$ relation. The second property follows from the first, when we restrict Γ to the subset which involves only name sorts. $\square$

The fundamental correctness property of αML is that two trees g and g' are α-equivalent iff their αML representations $[\![g]\!]$ and $[\![g']\!]$ are contextually equivalent. The two directions of this proof will be developed over the following two subsections.

5.4.3 α-equivalent ground trees are contextually equivalent

Given the definition of $[\![g]\!]$ in Definition 5.4.2, we can dispense with this direction of the proof relatively straightforwardly, by a general argument on the nature of αML expressions which correspond to translated trees.

Lemma 5.4.5 ($=_\alpha$ implies $\cong$). *If* $g =_\alpha g' : E$ *and* $\eta \vdash FN(g, g')$ *then* $\eta \vdash [\![g]\!] \cong [\![g']\!] : E$.

Proof. If $g =_\alpha g' : E$ then g and g' differ only by an α-renaming of their abstracted variables. However, in the translations $[\![g]\!]$ and $[\![g']\!]$ these variables are bound by $\exists$-quantifiers. Since we identify αML expressions up to α-conversion, it follows that $[\![g]\!]$ and $[\![g']\!]$ are in fact the same expression. Then, by Lemma 5.4.4 we have $\eta \vdash [\![g]\!] : E$, and since $\cong$ is reflexive it follows that $\eta \vdash [\![g]\!] \cong [\![g']\!] : E$, as required. $\square$

5.4.4 Contextually-equivalent ground trees are α-equivalent

This direction of the proof is less straightforward, because names n do not appear in the syntax of αML and because the representation $[\![g]\!]$ of a ground tree in αML is an expression, not a value. However, the proof is not as complicated as that for FreshML. This is because αML has built-in constructs for solving equality constraints, which actually involves checking whether values are α-equivalent. Therefore, if we know that two translated trees are contextually equivalent, then we know something about their operational behaviour when placed in contexts that are capable of testing α-equivalence. The following proof will exploit this capability of the αML meta-language.

Definition 5.4.6. Given a set of names $\overline{n}$, a name environment ε with $dom(\varepsilon) \supseteq \overline{n}$ and a valuation $V \in \alpha\text{-}Tree_\Sigma(\eta_\varepsilon)$ such that $V \models \#_{\varepsilon(\overline{n})}$, we write $\pi_{V,\varepsilon,\overline{n}}$ to stand for some (equivalently, any) permutation such that $V(\varepsilon(n)) = \{\pi_{V,\varepsilon,\overline{n}}(n)\}$ for all $n \in \overline{n}$. $\diamond$

Since names are not present in the syntax of αML, we will relate an αML expression $[\![g]\!]$ with the original ground tree g *up to a permutation*. This is related to the idea of unifying up to a permutation from equivariant unification (Cheney, 2005a). Using Definition 5.4.6, we will prove intermediate results about the relationship between a tree g and the constraints $\overline{c}_g$ and value v_g produced by evaluating its αML representation $[\![g]\!]$. Roughly, we will show that if a valuation V is injective on the variables corresponding to the free names of g then it is possible to permute the free names of g to produce a tree which is in the α-equivalence class produced when V is applied to the αML value v_g. As a shorthand we will write $\#_\varepsilon$ for the mutual freshness constraints $\#_{cod(\varepsilon)}$ (which is equivalent to $\#_{dom(\eta_\varepsilon)}$).

We now show that the result of evaluating a tree expression is equivalent (up to a permutation) to the tree itself. This is a central lemma in relating the operational behaviour of translated trees to their α-equivalence.

Lemma 5.4.7. *If $\varepsilon \vdash \langle \eta, g\rangle \triangleright \langle \eta', v_g\rangle$ then, for all $V \in \alpha\text{-}Tree_\Sigma(\eta')$, if $V \models \#_{dom(\eta')}$ then for some (or any) permutation $\pi_{V,\varepsilon,\overline{n}}$ it is the case that $(\pi_{V,\varepsilon,\overline{n}} \cdot g) \in [\![v_g]\!]_V$.*

Proof. The proof is by induction on the structure of g. We note that the assumption that $V \models \#_{dom(\eta')}$ is needed so that the permutation $\pi_{V,\varepsilon,\overline{n}}$ exists (see Definition 5.4.6). We present only the cases involving names: the cases for unit and data trees are straightforward and the tuple case relies on the fact that $\eta' \supseteq \eta$.

Base case: $g = n$. We assume that $dom(\varepsilon) = \overline{n}, \overline{n} \supseteq FN(n)$, $\eta \vdash_\varepsilon \overline{n}$ and $\varepsilon \vdash \langle \eta, n\rangle \triangleright \langle \eta', v_g\rangle$, and also that $V \in \alpha\text{-}Tree_\Sigma(\eta')$ and $V \models \#_{dom(\eta')}$. It follows that $n \in \overline{n}$, and also that $\eta' = \eta$ and hence that $n \in dom(\varepsilon)$. Therefore we know that $\varepsilon(n) = x$ and $v_g = x$, for some variable x. By definition of $\pi_{V,\varepsilon,\overline{n}}$ it follows that $V(\varepsilon(n)) = \{\pi_{V,\varepsilon,\overline{n}}(n)\}$, since $n \in \overline{n}$. Therefore we have that $[\![v_g]\!]_V = \{\pi_{V,\varepsilon,\overline{n}}(n)\}$, i.e. that $(\pi_{V,\varepsilon,\overline{n}} \cdot g) \in [\![v_g]\!]_V$, as required.

Inductive step: $g = <n>g'$. We assume that $dom(\varepsilon) = \overline{n}$, $\overline{n} \supseteq FN(<n>g')$, $\eta \vdash_\varepsilon \overline{n}$ and $\varepsilon \vdash \langle \eta, <n>g'\rangle \triangleright \langle \eta', v_g\rangle$, and also that $V \in \alpha\text{-}Tree_\Sigma(\eta')$ and $V \models \#_{dom(\eta')}$. It follows that $v_g = <x>v_{g'}$ where $x \notin dom(\eta)$, and that $\varepsilon[n \mapsto x] \vdash \langle \eta^*, g'\rangle \triangleright \langle \eta', v_{g'}\rangle$ where $\eta^* = \eta \cup \{x : sort(n)\}$. We can then show that $dom(\varepsilon[n \mapsto x]) = \overline{n} \cup \{n\}$, $\overline{n} \cup \{n\} \supseteq FN(g')$ and $\eta^* \vdash_{\varepsilon[n\mapsto x]} \overline{n} \cup \{n\}$ all hold. Then, by induction we get that $(\pi_{V,\varepsilon[n\mapsto x],\overline{n}\cup\{n\}} \cdot g') \in [\![v_{g'}]\!]_V$.

Now, if we write n^* for $\pi_{V,\varepsilon[n\mapsto x],\overline{n}\cup\{n\}}(n)$ then we have $<n^*>(\pi_{V,\varepsilon[n\mapsto x],\overline{n}\cup\{n\}} \cdot g') \in [<n^*>g^*]_\alpha$, where $g^* \in [\![v_{g'}]\!]_V$. By the definition of n^*, this becomes $(\pi_{V,\varepsilon[n\mapsto x],\overline{n}\cup\{n\}} \cdot <n>g') \in [<n^*>g^*]_\alpha$. Also, since $n \notin FN(<n>g')$ it follows that $(\pi_{V,\varepsilon[n\mapsto x],\overline{n}\cup\{n\}} \cdot <n>g') =_\alpha (\pi_{V,\varepsilon,\overline{n}} \cdot <n>g') : E$, for some equality type E: it does not matter whether the permutations agree on n since it is not free in the ground tree (Urban et al., 2004, Lemma 2.8). Therefore we have that $(\pi_{V,\varepsilon,\overline{n}} \cdot g) \in [<n^*>g^*]_\alpha$. Now, by definition of $\pi_{V,\varepsilon[n\mapsto x],\overline{n}\cup\{n\}}$ we get that $V(x) = \{n^*\}$. By Lemma 2.3.4 we get that $<n^*>g^* \in [\![<x>v_{g'}]\!]_V$. Therefore we have shown that $(\pi_{V,\varepsilon,\overline{n}} \cdot g) \in [\![v_g]\!]_V$ holds, as required.

This completes the proof of Lemma 5.4.7. $\square$

Lemma 5.4.8. *If $\emptyset \vdash \exists\Delta(\overline{c}; F; [\![g]\!]) : T$ and $\models \exists\Delta(\overline{c}\ \&\ \#_{dom(\eta_{\varepsilon_g})})$ both hold then $\exists\Delta(\overline{c}; F; [\![g]\!]) \longrightarrow \cdots \longrightarrow \exists\Delta, \eta_g(\overline{c}\ \&\ \overline{c}_g; F; v_g)$ and, for all $V \in \alpha\text{-}Tree_\Sigma(\Delta, \eta_g)$, if $V \models (\overline{c}\ \&\ \overline{c}_g)$ then for some (or any) permutation $\pi_{V,\varepsilon_g,(FN(g))}$ it is the case that $(\pi_{V,\varepsilon_g,(FN(g))} \cdot g) \in [\![v_g]\!]_V$.*

Proof. We assume that $\vdash \exists\Delta(\overline{c}; F; [\![g]\!]) : T$ and $\models \exists\Delta(\overline{c} \,\&\, \#_{dom(\eta_{\varepsilon_g})})$. By definition we know that $[\![g]\!]$ is $\exists(\eta' - \eta_{\varepsilon_g}).\,\#_{dom(\eta')} \,\&\, v_g$, where $\varepsilon_g \vdash \langle \eta_{\varepsilon_g}, g\rangle \,\triangleright\, \langle \eta', v_g\rangle$ holds. Writing η_g for $\eta' - \eta_{\varepsilon_g}$, from our first assumption we get that $\eta_{\varepsilon_g} \subseteq \Delta$, and by α-renaming we may assume that the newly-generated variables in $dom(\eta_g)$ are disjoint from $dom(\Delta)$. Therefore we may deduce that $\models \exists\Delta, \eta_g(\overline{c} \,\&\, \#_{dom(\eta')})$ holds, using our second assumption and the fact that the freshnesses involving the newly-generated variables in $dom(\eta_g)$ must be satisfiable. Hence we know that

$$\exists\Delta(\overline{c}; F; [\![g]\!]) \longrightarrow \cdots \longrightarrow \exists\Delta, \eta_g(\overline{c} \,\&\, \overline{c}_g; F; v_g)$$

holds, where $\overline{c}_g = \#_{dom(\eta')}$. Now we assume that $V \in \alpha\text{-}Tree_\Sigma(\eta', \eta_g)$ and $V \models (\overline{c} \,\&\, \overline{c}_g)$ both hold. Then, by Lemma 5.4.7 we get that $(\pi_{V,\varepsilon_g,(FN(g))} \cdot g) \in [\![v_g]\!]_V$ holds, for some (or any) permutation $\pi_{V,\varepsilon_g,FN(g)}$, as required. □

We can now use Lemma 5.4.8 to prove the main result of this section.

Lemma 5.4.9 ($\cong$ implies $=_\alpha$). *If $\eta \vdash [\![g]\!] \cong [\![g']\!] : E$ and $\eta \vdash FN(g, g')$ then $g =_\alpha g' : E$.*

Proof. We assume that $\eta \vdash [\![g]\!] \cong [\![g']\!] : E$ holds. This means that $[\![g]\!]$ and $[\![g']\!]$ have the same termination behaviour in any well-typed context. Writing ε^* for the result of merging the environments ε_g and $\varepsilon_{g'}$, we observe that $cod(\varepsilon^*) \subseteq dom(\eta)$. We now focus on the behaviour of the following two configurations.

1. $\exists\eta(\#_{\varepsilon^*}; \mathtt{Id} \circ (z.\,\mathtt{let}\, y = [\![g']\!] \,\mathtt{in}\, z = y); [\![g']\!])$
2. $\exists\eta(\#_{\varepsilon^*}; \mathtt{Id} \circ (z.\,\mathtt{let}\, y = [\![g']\!] \,\mathtt{in}\, z = y); [\![g]\!])$

From contextual equivalence we know that the termination behaviour of configurations 1 and 2 are identical. Thus it suffices to show firstly that configuration 1 terminates, and secondly that if configuration 2 terminates then $g =_\alpha g' : E$ holds. The proofs of these follow.

- **Configuration 1 terminates.**
  Since $\eta \vdash FN(g, g')$ we know that $\models \exists\eta(\#_{\varepsilon^*})$ holds. Therefore, by Lemma 5.4.8 we know that

$$\begin{aligned}&\exists\eta(\#_{\varepsilon^*}; \mathtt{Id} \circ (z.\,\mathtt{let}\, y = [\![g']\!] \,\mathtt{in}\, z = y); [\![g']\!])\\ &\qquad \longrightarrow \cdots \longrightarrow \exists\eta, \eta_1(\#_{\varepsilon^*} \,\&\, \overline{c}_1; \mathtt{Id} \circ (z.\, v_1 = z); [\![g']\!])\end{aligned}$$

  holds, after some constraint simplification. By preservation of satisfiability (Theorem 3.7.6) we also know that the new constraint is satisfiable. By applying Lemma 5.4.8 again to the reduced configuration we get that

$$\begin{aligned}&\exists\eta, \eta_1(\#_{\varepsilon^*} \,\&\, \overline{c}_1; \mathtt{Id} \circ (z.\, v_1 = z); [\![g']\!])\\ &\qquad \longrightarrow \cdots \longrightarrow \exists\eta, \eta_1, \eta_2(\#_{\varepsilon^*} \,\&\, \overline{c}_1 \,\&\, \overline{c}_2; \mathtt{Id}; v_1 = v_2)\end{aligned}$$

  holds (also after some constraint simplification). We also know that there exists a valuation $V \in \alpha\text{-}Tree_\Sigma(\eta, \eta_1, \eta_2)$ such that $V \models (\#_{\varepsilon^*} \,\&\, \overline{c}_1 \,\&\, \overline{c}_2)$ and $(\pi_{V,\varepsilon^*,(FN(g'))} \cdot g') \in [\![v_2]\!]_V$ both hold. Furthermore, since $V \models \#_{\varepsilon^*} \,\&\, \overline{c}_1$ we also know that $(\pi_{V,\varepsilon^*,(FN(g'))} \cdot g') \in [\![v_1]\!]_V$. Now, in order to show that configuration 1 terminates, it suffices to show that

$$\models \exists\eta, \eta_1, \eta_2(\#_{\varepsilon^*} \,\&\, \overline{c}_1 \,\&\, \overline{c}_2 \,\&\, v_1 = v_2). \tag{5.4}$$

  We have already shown that $V \models \#_{\varepsilon^*} \,\&\, \overline{c}_1 \,\&\, \overline{c}_2$, so it remains only to see that $V \models v_1 = v_2$. Since $(\pi_{V,\varepsilon^*,(FN(g'))} \cdot g') =_\alpha (\pi_{V,\varepsilon^*,(FN(g'))} \cdot g') : E$ holds by reflexivity, it follows that $[\![v_1]\!]_V = [\![v_2]\!]_V$, and hence that $V \models v_1 = v_2$. Thus we have shown that (5.4) holds, and hence we conclude that configuration 1 terminates.

- **If configuration 2 terminates then $g =_\alpha g' : E$.**
  We assume that configuration 2 terminates. Thus we know that

$$\begin{array}{rl} & \exists\eta(\#_{\varepsilon^*};\texttt{Id}\circ(z.\,\texttt{let}\,y = [\![g']\!]\ \texttt{in}\ z = y);[\![g]\!]) \\ \longrightarrow\cdots\longrightarrow & \exists\eta,\eta_g(\#_{\varepsilon^*}\ \&\ \overline{c}_g;\texttt{Id}\circ(z.\,v_g = z);[\![g']\!]) \\ \longrightarrow\cdots\longrightarrow & \exists\eta,\eta_g,\eta_{g'}(\#_{\varepsilon^*}\ \&\ \overline{c}_g\ \&\ \overline{c}_{g'};\texttt{Id};v_g = v_{g'}) \\ \longrightarrow & \exists\eta,\eta_g,\eta_{g'}(\#_{\varepsilon^*}\ \&\ \overline{c}_g\ \&\ \overline{c}_{g'}\ \&\ v_g = v_{g'};\texttt{Id};\top) \end{array}$$

  after some constraint simplification, where η_g, $\overline{c}_g$ and v_g are the results of evaluating $[\![g]\!]$ and similarly $\eta_{g'}$, $\overline{c}_{g'}$ and $v_{g'}$ were produced by evaluation of $[\![g']\!]$. Furthermore, we know that the final constraint is satisfiable, i.e. that

$$\models \exists\eta,\eta_g,\eta_{g'}(\#_{\varepsilon^*}\ \&\ \overline{c}_g\ \&\ \overline{c}_{g'}\ \&\ v_g = v_{g'}).$$

  From this we get that there exists $V \in \alpha\text{-}Tree_\Sigma(\eta,\eta_g,\eta_{g'})$ such that $V \models \#_{\varepsilon^*}$ and $[\![v_g]\!]_V = [\![v_{g'}]\!]_V$ both hold (as well as $V \models \overline{c}_g$ and $V \models \overline{c}_{g'}$). Then, by Lemma 5.4.8 we get that $(\pi_{V,\varepsilon^*,(FN(g))}\cdot g) \in [\![v_g]\!]_V$ and $(\pi_{V,\varepsilon^*,(FN(g'))}\cdot g') \in [\![v_{g'}]\!]_V$ both hold, for some/any permutations $\pi_{V,\varepsilon^*,(FN(g))}$ and $\pi_{V,\varepsilon^*,(FN(g'))}$. Since $[\![v_g]\!]_V = [\![v_{g'}]\!]_V$, it follows that

$$(\pi_{V,\varepsilon^*,(FN(g))}\cdot g) =_\alpha (\pi_{V,\varepsilon^*,(FN(g'))}\cdot g') : E. \tag{5.5}$$

  Now, since $V \models \#_{\varepsilon^*}$, we consider permutations of the form $\pi_{V,\varepsilon^*,\overline{n}^*}$, where $\overline{n}^* = dom(\varepsilon^*)$. By definition of ε^*, for some/any such permutation it is the case that

$$\begin{array}{c} \forall n \in FN(g).\ \pi_{V,\varepsilon^*,\overline{n}^*}(n) = \pi_{V,\varepsilon^*,(FN(g))}(n) \\ \forall n \in FN(g').\ \pi_{V,\varepsilon^*,\overline{n}^*}(n) = \pi_{V,\varepsilon^*,(FN(g'))}(n) \end{array}$$

  both hold, and by (Urban et al., 2004, Lemma 2.8) it follows that

$$\begin{array}{c} (\pi_{V,\varepsilon^*,\overline{n}^*}\cdot g) =_\alpha (\pi_{V,\varepsilon^*,(FN(g))}\cdot g) : E \\ (\pi_{V,\varepsilon^*,\overline{n}^*}\cdot g') =_\alpha (\pi_{V,\varepsilon^*,(FN(g'))}\cdot g') : E \end{array}$$

  which may be combined with (5.5) to yield $(\pi_{V,\varepsilon^*,\overline{n}^*}\cdot g) =_\alpha (\pi_{V,\varepsilon^*,\overline{n}^*}\cdot g') : E$. Finally, by (Urban et al., 2004, equation 9) we can eliminate the permutations from both sides to leave $g =_\alpha g' : E$, as required.

This completes the proof of Lemma 5.4.9. □

5.4.5 Fundamental correctness property

The fundamental correctness property of αML follows from Lemma 5.4.5 and Lemma 5.4.9.

Theorem 5.4.10 (Fundamental correctness property). *If $\eta \vdash FN(g, g')$ then $g =_\alpha g' : E$ holds if and only if $\eta \vdash [\![g]\!] \cong [\![g']\!] : E$.* □

This result is interesting because our representation of ground trees is correct up to α-equivalence but does not rely on the existence of globally-fresh names. These are central to similar encodings of λ-terms into FreshML—see, for example, (Shinwell, 2005, Definition 3.7.2).

5.5 Contextual equivalence of formulae

Recalling the encodings of schematic formulae and α-inductive definitions as αML expressions, from Chapter 4, the *logical soundness and completeness* result (Theorem 4.3.2) for encoded formulae gives us a weak result about the operational equivalence of encoded formulae:

Corollary 5.5.1 (Equivalence for CLP goal states). *If $\mathcal{D} \models \forall\Delta.\, \varphi \equiv \varphi'$ then*

$$\exists\Delta'(\overline{c}; F_{\vec{\varphi}}; \varphi[v_{\mathcal{D}}/r])\!\downarrow \iff \exists\Delta'(\overline{c}; F_{\vec{\varphi}}; \varphi'[v_{\mathcal{D}}/r])\!\downarrow$$

for any $\Delta' \supseteq \Delta$, any $\overline{c}$ and any frame stack $F_{\vec{\varphi}}$ which corresponds to a CLP goal list in the sense of Definition 4.2.1. □

If the formulae φ and φ' have the same semantics then $\overline{c}\ \&\ \vec{\varphi}\ \&\ \varphi$ and $\overline{c}\ \&\ \vec{\varphi}\ \&\ \varphi'$ also have the same semantics. Then, Corollary 5.5.1 follows straightforwardly from Theorem 4.3.2. However, there is a more general result about the relationship between semantic equivalence of schematic formulae and contextual equivalence of the encoded formulae in *arbitrary* αML contexts.

Theorem 5.5.2. *For all $\mathcal{D}$, $\mathcal{D}'$, Δ, φ and φ' it is the case that $\Delta \vdash \varphi[v_{\mathcal{D}}/r] \cong \varphi'[v_{\mathcal{D}'}/r] : \mathtt{prop}$ iff $\mathcal{D}, \mathcal{D}' \models \forall\Delta.\, \varphi \equiv \varphi'$.* □

This result is interesting because the full range of contexts in αML are richer than just CLP goal states—in particular, the presence of higher-order functions means that formulae may be packaged up inside a function and passed around before eventually being evaluated. In this setting it is by no means obvious that semantically equivalent formulae always have the same behaviour with regard to termination. Furthermore this result relates the behaviour of formulae which are equivalent but with regard to different α-inductive definitions. The proof of Theorem 5.5.2 appears as Appendix B.

5.6 Operational equivalence with finite failure

The motivations for studying $\cong^\circ$ are its simplicity and certain attractive theoretical results, such as its coincidence with α-equivalence for encoded ground trees (Theorem 5.4.10) and with semantic equivalence for schematic formulae (Theorem 5.5.2). However, an expression which always diverges and an expression which always fails finitely are indistinguishable under $\cong^\circ$, even though they could probably be told apart by evaluating them in an αML interpreter. In this section we discuss a version of operational equivalence which allows both successful termination and finite failure to be observed. We also explore the relationship between this notion of program equivalence and the coarser one defined in Chapter 5.

Definition 5.6.1 (Operational equivalence with finite failure). Recalling the definitions of success (Definition 3.7.1) and finite failure (Definition 3.7.2), we define another operational equivalence relation $\boldsymbol{\Delta \vdash e \cong_F e' : T}$ which holds iff

- $\Delta \vdash e : T$ and $\Delta \vdash e' : T$; and
- for all Δ', $\overline{c}$, F and T' such that $\Delta' \supseteq \Delta$ and $\Delta' \vdash \overline{c} : \mathtt{prop}$ and $\Delta' \vdash F : T \to T'$, the following equivalences both hold:

$$\exists\Delta'(\overline{c}; F; e)\!\downarrow \iff \exists\Delta'(\overline{c}; F; e')\!\downarrow, \tag{5.6}$$

$$\exists\Delta'(\overline{c}; F; e)\ \mathit{fails} \iff \exists\Delta'(\overline{c}; F; e')\ \mathit{fails}. \tag{5.7}$$

◇

We extend this definition to a relation $\cong_F^\circ$ between "open" expressions (i.e. those containing variables which are not of an equality type) in precisely the same way that $\cong^\circ$ is defined in terms of $\cong$ (see Definition 5.1.2). The following theorem relates $\cong_F^\circ$ to the original definition of $\cong^\circ$ (a similar relationship can be demonstrated between $\cong_F$ and $\cong$).

Theorem 5.6.2 (Relationship to $\cong^\circ$). *For all Γ, e, e', T, it is the case that $\Gamma \vdash e \cong_F^\circ e' : T$ implies $\Gamma \vdash e \cong^\circ e' : T$.*

Proof. This is a straightforward consequence of the fact that the definition of $\cong_F$ (and therefore $\cong_F^\circ$) is identical to that of $\cong$ except for the extra condition on finite failure behaviour. Therefore, if two expressions are $\cong_F^\circ$-related then they certainly have identical termination behaviour, which means that they are also $\cong^\circ$-related. $\square$

Note that the reverse implication does not hold, because $\cong^\circ$ does not distinguish between failing and divergent computations. However, we can test for the possibility of divergence using $\cong_F^\circ$. We begin by defining type-indexed divergent expressions (Ω_T) and failing expressions (χ_T) as follows.

$$\Omega_T \triangleq (\mathtt{fun}\, f\,(x : ()) : T = f\; x)\; () \tag{5.8}$$
$$\chi_T \triangleq \exists x : N.\, (x \mathbin{\#} x) \mathbin{\&} \Omega_T. \tag{5.9}$$

It is clear that Ω_T will reduce forever in any configuration and that χ_T will fail finitely in any configuration. We can now provide a counter-example to the converse of Theorem 5.6.2. Since $\Gamma \vdash \Omega_T : T$ and $\Gamma \vdash \chi_T : T$ both hold, we have that $\Gamma \vdash \chi_T \cong^\circ \Omega_T : T$ holds, because neither expression ever terminates successfully. However, $\Gamma \vdash \chi_T \cong_F^\circ \Omega_T : T$ does not hold because χ_T always fails finitely whereas Ω_T always diverges. Therefore, their operational behaviour with respect to finite failure is different.

We have proven a CIU theorem for $\cong_F^\circ$, along the lines of Theorem 5.3.1. The proofs are along the same lines as those from Section 5.3 so we omit the details in the interest of brevity. The arguments for failure behaviour are almost the same as for termination, except in the cases where non-determinism or finite failure are a possibility.

We can also extend the fundamental correctness theorem of αML (which pertains to the encoding of ground trees modulo α-equivalence) to the finer $\cong_F$ relation, as follows.

Theorem 5.6.3 (Fundamental correctness property for $\cong_F^\circ$). *If $\eta \vdash FN(g, g')$ then $g =_\alpha g' : E$ holds iff $\eta \vdash [\![g]\!] \cong_F [\![g']\!] : E$ holds.*

Proof. The forward direction holds because, as in the proof of Lemma 5.4.5, $[\![g]\!]$ and $[\![g']\!]$ are the same expression in αML, since we identify expressions up to α-renaming. Then, the result follows because $\cong_F$ is reflexive. The reverse implication holds because any pair of expressions which are $\cong_F^\circ$-related must also be $\cong^\circ$-related, and the result follows by Lemma 5.4.9. $\square$

Allowing us to observe finite failure in addition to successful termination does not affect the correctness of the representation of ground trees. Intuitively, this is because the evaluation of such expressions never diverges, so observing successful termination is equivalent to observing both successful termination and finite failure.

Finally, we consider the properties of $\cong_F^\circ$ with respect to semantic equivalence of schematic formulae. As above, it is straightforward to show that if two formulae are $\cong_F$-related then they have the same semantics.

Theorem 5.6.4. *For all $\mathcal{D}$, $\mathcal{D}'$, Δ, φ and φ', if $\Delta \vdash \varphi[v_{\mathcal{D}}/r] \cong_F \varphi'[v_{\mathcal{D}'}/r] : \mathtt{prop}$ then $\mathcal{D}, \mathcal{D}' \models \forall \Delta.\, \varphi \equiv \varphi'$.*

Proof. We assume that $\Delta \vdash \varphi[v_{\mathcal{D}}/r] \cong_F \varphi'[v_{\mathcal{D}'}/r] : \texttt{prop}$. Then, by Theorem 5.6.2 we get that $\Delta \vdash \varphi[v_{\mathcal{D}}/r] \cong \varphi'[v_{\mathcal{D}'}/r] : \texttt{prop}$ holds, and then the result follows by Theorem 5.5.2. $\square$

The reverse implication does not hold—pairs of formulae exist which have the same semantics but have different operational behaviour with respect to finite failure. To demonstrate this we will construct a suitable inductive definition and exhibit the counterexample as formulae over that definition. Following the definition of α-inductive definitions in Chapter 2, we assume that the datatype S_r has two constructors:

$$\texttt{datatype}\ S_r =_\Sigma K_1\ \texttt{of}\ N \mid K_2\ \texttt{of}\ N$$

where N is a name sort in $\mathbb{N}_\Sigma$. We will use a single relation symbol r to create the inductive definition $\mathcal{D}^*$, which is specified by the following two rules.

$$\frac{y \mathrel{\#} y}{r\,(K_1\,y)} \qquad\qquad \frac{r\,(K_2\,y)}{r\,(K_2\,y)}$$

Following Definition 2.4.2, we can translate $\mathcal{D}^*$ into the following standard form.

$$\frac{(\exists y : N.\ x = (K_1\,y)\ \&\ y \mathrel{\#} y) \vee (\exists y : N.\ x = (K_2\,y)\ \&\ r\,(K_2\,y))}{r\,x}$$

We now define two schematic formulae

$$\varphi_1 \triangleq r\,(K_1\,y) \qquad\qquad \varphi_2 \triangleq r\,(K_2\,y)$$

and consider their semantics. It is not difficult to see that neither $\exists V.\ V \models \varphi_1$ nor $\exists V.\ V \models \varphi_2$ holds—in the first instance because the freshness constraint $y \mathrel{\#} y$ is never satisfiable and in the second because the definition of $[\![\mathcal{D}^*]\!]$ as a least fixed point means that any circular definition has an empty denotation. Therefore, we have

$$\mathcal{D}^* \models \forall y : N.\ \varphi_1 \equiv \varphi_2.$$

However, the operational equivalence judgement $y : N \vdash \varphi_1[v_{\mathcal{D}^*}/r] \cong_F \varphi_2[v_{\mathcal{D}^*}/r] : \texttt{prop}$ does not hold, because for any (suitably well-typed) configuration $\exists\Delta(\overline{c}; F; -)$ it is the case that $\exists\Delta(\overline{c}; F; \varphi_1[v_{\mathcal{D}^*}/r])$ fails finitely whereas $\exists\Delta(\overline{c}; F; \varphi_2[v_{\mathcal{D}^*}/r])$ diverges.

This is not surprising given that the denotation of a schematic inductive definition is a recursively enumerable set. This means that we could encode a universal Turing machine as an α-inductive definition. If the reverse containment of semantic equivalence within $\cong_F^\circ$ were to hold, we could decide the Halting Problem (Turing, 1936) by simulating a universal Turing machine in αML and checking for success or finite failure.

5.7 Fresh name generation

We conclude this chapter by considering a key feature of many nominal meta-programming languages which is absent from αML: fresh name generation.

Existing languages such as FreshML provide support for generating new names which are implicitly globally fresh. For example, (Shinwell, 2005, Figure 3.4) defines a big-step semantics for FreshML with judgements of the form $\overline{n}, e \Downarrow v, \overline{n}'$. The sets $\overline{n}$ and $\overline{n}'$ retain the "state" of generated names, which obey the permutative convention. That figure contains the following generating fresh names (paraphrased slightly).

$$\frac{}{\overline{n}, \texttt{fresh} \Downarrow n, \overline{n} \uplus \{n\}}\ n \in \mathit{Name} - \overline{n}$$

Name is the countably infinite set of names, and the rule works by simply selecting any name which does not appear in the (finite) set of names $\overline{n}$ that have already been chosen.

This contrasts sharply with the αML design philosophy of using meta-variables (which may be aliased) to represent bindable names. In the encoding of ground trees described in Section 5.4.2, for example, explicit freshness constraints are required to ensure that the meta-variables used to represent the names in the tree are distinct from each other. This approach works because the names only need to be locally fresh for Theorem 5.4.10 to be true. However, it is difficult to see how explicit freshness constraints could be used to model dynamic fresh name generation.

In order to express this idiom in αML, we will demonstrate an extension to the core language defined in Section 3.2 which allows us to mimic the fresh name generation facilities of FreshML and αProlog. We extend the grammar of expressions with a new construct for generating a globally fresh name of a particular name sort N:

$$e \quad ::= \quad \cdots \mid \texttt{fresh}\, N.$$

The type annotation is required in order to make type inference trivial: the typing rule (and compatible refinement rule) for this term-former are straightforward.

$$\frac{}{\Gamma \vdash \texttt{fresh}\, N : N} \qquad\qquad \frac{}{\Gamma \vdash (\texttt{fresh}\, N)\, \widehat{\mathcal{E}}\, (\texttt{fresh}\, N) : N}$$

Finally, we extend the operational semantics of αML (Figure 3.4) with an additional impure reduction rule to handle `fresh`-expressions.

(I7) $\exists \Delta(\overline{c}; F; \texttt{fresh}\, N) \longrightarrow \exists \Delta, x : N(\overline{c}\ \&\ x \,\#\, \Delta; F; x)$ where $x \notin dom(\Delta)$.

In the definition of rule (I7) we write $x \,\#\, \Delta$ for the constraint $x \,\#\, x_1\ \&\ \cdots\ \&\ x \,\#\, x_n$, where $dom(\Delta) = \{x_1, \ldots, x_n\}$. The fresh name is returned as a newly-generated meta-variable, together with a set of new freshness constraints which require it to be fresh for every other meta-variable generated so far. These extra constraints make explicit the implicit convention of FreshML that distinct names are fresh for each other, and also that the newly-generated name may not be free in any unknown object-level data terms. This is analogous to the extra freshness constraints added to the environment during the evaluation of an αProlog goal containing a fresh name quantifier И (Cheney and Urban, 2008, Figure 13). These changes are relatively straightforward, and show that fresh name generation can be added cleanly to αML without abandoning the "names as meta-variables" design philosophy.

We can now use the `fresh` construct to define a "generative unbinding" operator (Pitts and Shinwell, 2008) which mimics that of FreshML:

$$\begin{aligned} &\texttt{unbind_fresh}\, e\ \texttt{as}\ \texttt{<}x\texttt{>}x' : [N]E\ \texttt{in}\ e' \triangleq \\ &\qquad \texttt{let}\ x = \texttt{fresh}\, N\ \texttt{in}\ \exists x' : E.\, (\texttt{<}x\texttt{>}x' = e)\ \&\ e' \end{aligned} \tag{5.10}$$

The definition of `unbind_fresh` from (5.10), along with the extra operational rule (I7), produces a version of generative unbinding similar to that used in MLSOS (Lakin and Pitts, 2008). However, its operational behaviour is very different to the unbinding facilities of FreshML (Pitts and Shinwell, 2008; Shinwell, 2005). In αML, the `unbind_fresh` construct generates a globally fresh meta-variable x to stand for the unbound name and a new meta-variable to stand for the freshened abstraction body. These meta-variables constitute a pattern which is used in the equality constraint to perform pattern-matching up to α-equivalence.

The literature on FreshML tends not to include generative unbinding as a language primitive but rather defines it in terms of more fundamental operations such as fresh name generation and name-swapping—see (Shinwell et al., 2003; Shinwell and Pitts, 2005). A definition

of generative unbinding in FreshML (corresponding to our definition (5.10)) might look as follows.

$$\texttt{unbind_fresh}\ \texttt{<}x\texttt{>}v\ \texttt{as}\ \texttt{<}x'\texttt{>}x''\ \texttt{in}\ e' \triangleq$$
$$\texttt{let}\ x' = \texttt{fresh}\ N\ \texttt{in}\ \texttt{let}\ x'' = (\texttt{swap}\ x\,\texttt{,}\,x'\ \texttt{in}\ v)\ \texttt{in}\ e'$$

In this rule a fresh name x' is indeed generated, and is then swapped for the existing bound name throughout the body of the abstraction, using the `swap` operation. This action prevents capture by ensuring that bound names are freshened up before the programmer can access them directly. A naïve implementation of swapping involves walking the entire structure of the value v and applying the swapping to all names that are encountered: more efficient implementations apply the swapping lazily, pushing it through the outermost term constructor only when required (see (Shinwell, 2005, Section 6.6) for a detailed discussion of the pragmatic aspects of swapping).

It is not known whether the αML and FreshML approaches to generative unbinding are behaviourally equivalent at equality types. One thing, however, is clear: the αML version is more restricted in terms of the types of values that may be decomposed using `unbind_fresh`. Applying the αML typing rules to the definition (5.10) makes it clear that only expressions of type $\texttt{[}N\texttt{]}\,E$ can be unbound, where E must be an equality type. This restriction is essential in αML because nominal constraint solving is used to implement α-equivalent pattern matching, and constraints must be between values of equality types. The type system of FreshML is more liberal: one can generatively unbind values of any type, for example $\texttt{[}N\texttt{]}\,\texttt{(}T \to T'\texttt{)}$, by simply pushing the appropriate swapping through the run-time representation of the body of the abstraction, for example a function closure.

It is straightforward to rework the theory of contextual equivalence for αML developed in this chapter to apply to the extended language with fresh name generation. Since the concepts used to define the operational semantics of the `fresh` operator already exist in the αML meta-language, namely existential quantification and explicit freshness constraints, we may ask whether `fresh` is directly definable in terms of the constructs present in the core αML language from Figure 3.1.

Conjecture 5.7.1. `fresh` *is* not *directly definable in core αML.* ◇

It seems unlikely that `fresh` construct is definable in a compositional way. This is because the operational rule (I7) presented above requires run-time introspection of the state of the abstract machine, in order to determine the set of previously-generated variables $dom(\Delta)$. No such features are presented in core αML, so this information on the current abstract machine configuration would need to be provided explicitly to any program construct attempting to mimic its behaviour. This, in combination with the higher-order features of αML, suggests that it should be possible to construct two expressions that have different properties with respect to contexutal equivalence depending on whether `fresh` is permitted.

However, we have been unable to find such an example which would confirm this, but neither have we found a provably correct, compositional translation of the `fresh` construct into core αML. Therefore, Conjecture 5.7.1 remains an open research question.

Chapter 6

Constraint solving

> *"There are no constraints on the human mind ... except those we ourselves erect."*
>
> —R. Reagan

We introduced α-tree constraint problems and the decision problem NonPermSat in Section 3.4. We noted that NonPermSat is a subproblem of the equivariant unification problem considered by Cheney (Cheney, 2004b, Section 7.2.4), and used this fact to show that NonPermSat is decidable (Theorem 3.4.5). We also showed that it is NP-complete (Theorem 3.4.9). This fact told us that NonPermSat and equivariant unification are equivalent.

In this section we describe an algorithm which can solve NonPermSat directly in many cases. This is motivated by a desire to process the constraint problems that arise during the execution of αML programs without the additional machinery needed for full-blown equivariant unification.

6.1 Constraint transformation

We recall the definition of constraints and constraint problems from Section 3.4. We recall that the terms t that feature in non-permutative constraints are given by the grammar

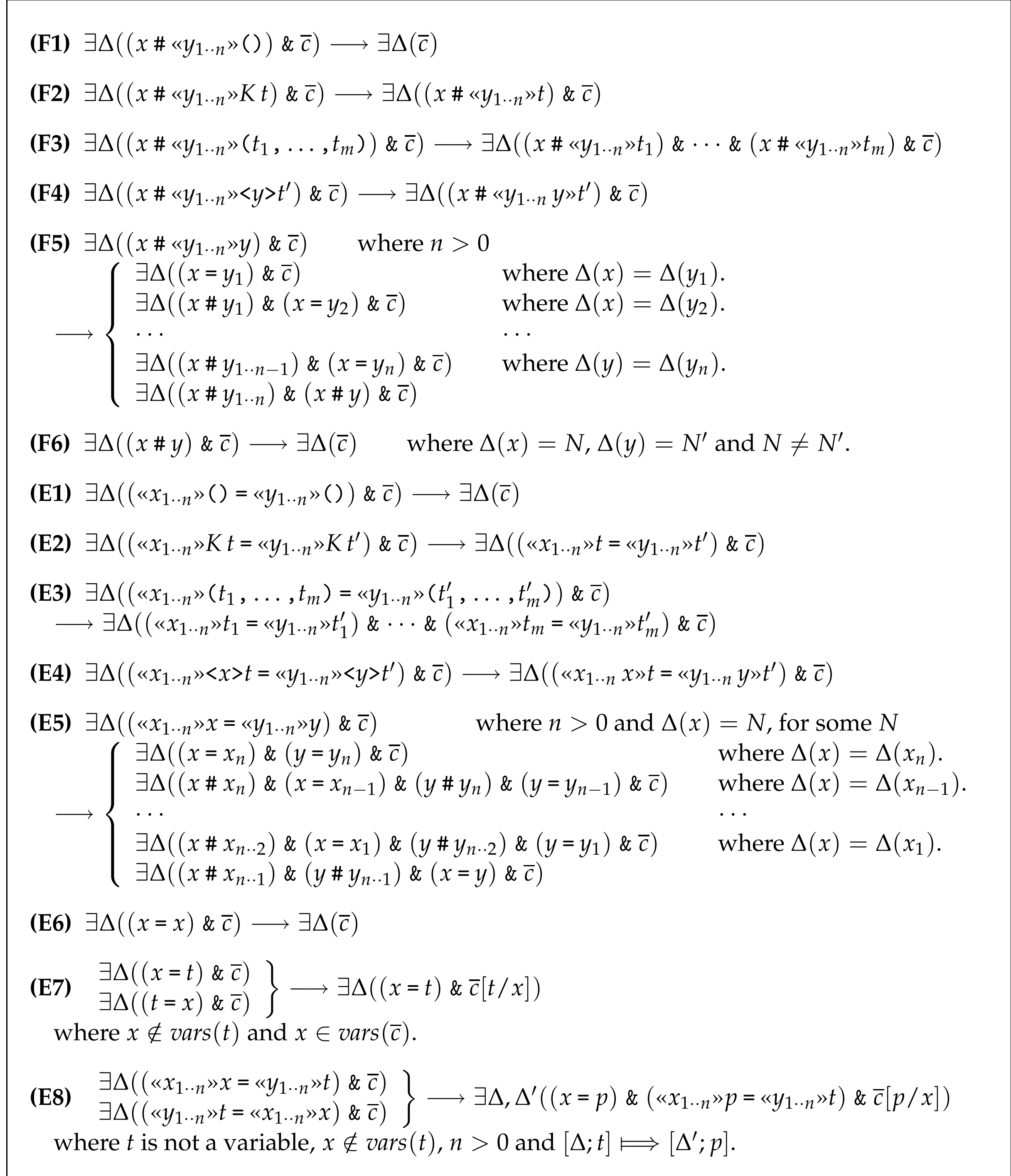
$$t \quad ::= \quad x \mid \texttt{<}x\texttt{>}t \mid K\,t \mid (t_1, \ldots, t_n) \mid ().$$

In this chapter we will consider non-permutative constraints of the forms

$$\begin{array}{rcll} c & ::= & «x_{1..n}»t = «y_{1..n}»t' & \text{(equality)} \\ & \mid & x \mathrel{\#} «y_{1..n}»t' & \text{(freshness).} \end{array}$$

The lists of variables $«x_{1..n}»$ are a presentational device intended to simplify the presentation of the transformation rules. They represent a context of nested abstractions at the head of the term. Unless explicitly stated otherwise, these lists may be of zero length. The semantics of constraints is as presented in Definition 3.4.3. For the purposes of semantics, the notation $«x_{1..n}»t$ should be thought of as representing $\texttt{<}x_1\texttt{>} \cdots \texttt{<}x_n\texttt{>}t$.

We define a non-deterministic transition relation $\longrightarrow$ which transforms a single constraint problem into a finite, non-empty set of constraint problems. Figure 6.1 presents transition rules for the $\longrightarrow$ transformation relation. To save space, we write $x \mathrel{\#} x_{1..n}$ for the conjunction $x \mathrel{\#} x_1 \mathrel{\&} \cdots \mathrel{\&} x \mathrel{\#} x_n$.



(F1) $\exists\Delta((x \# «y_{1..n}»()) \;\&\; \overline{c}) \longrightarrow \exists\Delta(\overline{c})$

(F2) $\exists\Delta((x \# «y_{1..n}»K\,t) \;\&\; \overline{c}) \longrightarrow \exists\Delta((x \# «y_{1..n}»t) \;\&\; \overline{c})$

(F3) $\exists\Delta((x \# «y_{1..n}»(t_1, \ldots, t_m)) \;\&\; \overline{c}) \longrightarrow \exists\Delta((x \# «y_{1..n}»t_1) \;\&\; \cdots \;\&\; (x \# «y_{1..n}»t_m) \;\&\; \overline{c})$

(F4) $\exists\Delta((x \# «y_{1..n}»\text{<}y\text{>}t') \;\&\; \overline{c}) \longrightarrow \exists\Delta((x \# «y_{1..n}\,y»t') \;\&\; \overline{c})$

(F5) $\exists\Delta((x \# «y_{1..n}»y) \;\&\; \overline{c})$ where $n > 0$

$$\longrightarrow \begin{cases} \exists\Delta((x = y_1) \;\&\; \overline{c}) & \text{where } \Delta(x) = \Delta(y_1). \\ \exists\Delta((x \# y_1) \;\&\; (x = y_2) \;\&\; \overline{c}) & \text{where } \Delta(x) = \Delta(y_2). \\ \ldots & \ldots \\ \exists\Delta((x \# y_{1..n-1}) \;\&\; (x = y_n) \;\&\; \overline{c}) & \text{where } \Delta(y) = \Delta(y_n). \\ \exists\Delta((x \# y_{1..n}) \;\&\; (x \# y) \;\&\; \overline{c}) & \end{cases}$$

(F6) $\exists\Delta((x \# y) \;\&\; \overline{c}) \longrightarrow \exists\Delta(\overline{c})$ where $\Delta(x) = N$, $\Delta(y) = N'$ and $N \neq N'$.

(E1) $\exists\Delta((«x_{1..n}»() = «y_{1..n}»()) \;\&\; \overline{c}) \longrightarrow \exists\Delta(\overline{c})$

(E2) $\exists\Delta((«x_{1..n}»K\,t = «y_{1..n}»K\,t') \;\&\; \overline{c}) \longrightarrow \exists\Delta((«x_{1..n}»t = «y_{1..n}»t') \;\&\; \overline{c})$

(E3) $\exists\Delta((«x_{1..n}»(t_1, \ldots, t_m) = «y_{1..n}»(t'_1, \ldots, t'_m)) \;\&\; \overline{c})$
$\longrightarrow \exists\Delta((«x_{1..n}»t_1 = «y_{1..n}»t'_1) \;\&\; \cdots \;\&\; («x_{1..n}»t_m = «y_{1..n}»t'_m) \;\&\; \overline{c})$

(E4) $\exists\Delta((«x_{1..n}»\text{<}x\text{>}t = «y_{1..n}»\text{<}y\text{>}t') \;\&\; \overline{c}) \longrightarrow \exists\Delta((«x_{1..n}\,x»t = «y_{1..n}\,y»t') \;\&\; \overline{c})$

(E5) $\exists\Delta((«x_{1..n}»x = «y_{1..n}»y) \;\&\; \overline{c})$ where $n > 0$ and $\Delta(x) = N$, for some N

$$\longrightarrow \begin{cases} \exists\Delta((x = x_n) \;\&\; (y = y_n) \;\&\; \overline{c}) & \text{where } \Delta(x) = \Delta(x_n). \\ \exists\Delta((x \# x_n) \;\&\; (x = x_{n-1}) \;\&\; (y \# y_n) \;\&\; (y = y_{n-1}) \;\&\; \overline{c}) & \text{where } \Delta(x) = \Delta(x_{n-1}). \\ \ldots & \ldots \\ \exists\Delta((x \# x_{n..2}) \;\&\; (x = x_1) \;\&\; (y \# y_{n..2}) \;\&\; (y = y_1) \;\&\; \overline{c}) & \text{where } \Delta(x) = \Delta(x_1). \\ \exists\Delta((x \# x_{n..1}) \;\&\; (y \# y_{n..1}) \;\&\; (x = y) \;\&\; \overline{c}) & \end{cases}$$

(E6) $\exists\Delta((x = x) \;\&\; \overline{c}) \longrightarrow \exists\Delta(\overline{c})$

(E7) $\left.\begin{array}{l} \exists\Delta((x = t) \;\&\; \overline{c}) \\ \exists\Delta((t = x) \;\&\; \overline{c}) \end{array}\right\} \longrightarrow \exists\Delta((x = t) \;\&\; \overline{c}[t/x])$
where $x \notin vars(t)$ and $x \in vars(\overline{c})$.

(E8) $\left.\begin{array}{l} \exists\Delta((«x_{1..n}»x = «y_{1..n}»t) \;\&\; \overline{c}) \\ \exists\Delta((«y_{1..n}»t = «x_{1..n}»x) \;\&\; \overline{c}) \end{array}\right\} \longrightarrow \exists\Delta, \Delta'((x = p) \;\&\; («x_{1..n}»p = «y_{1..n}»t) \;\&\; \overline{c}[p/x])$
where t is not a variable, $x \notin vars(t)$, $n > 0$ and $[\Delta; t] \Longrightarrow [\Delta'; p]$.

Figure 6.1: Constraint transformation rules

Rules (F1)–(F3) and (E1)–(E3) deal with unit, data and tuple terms in the usual way: the only difference is that we work within nested abstractions. The abstractions do not play any part in these six rules, except that the lists on both sides of an equality constraint must be of the same length. The rules (F4) and (E4) deal with abstractions by simply adding them to the "environment" at the front of the term. The rules (F6) and (E6) dispose of trivial constraints: in the case of (F6), two names of different sorts will always be fresh for each other and in the case of (E6), any term is equal to itself.

The most interesting rules are (F5) and (E5), which deal with the scopes of bound names with respect to the nested abstractions. We first consider (F5). In order for x to not appear free anywhere in «$y_{1..n}$»y, either x should map to the same name as one of the abstracted variables $y_1, \ldots, y_n$ or x should be distinct from all of the abstracted variables and be constrained to be fresh for the unknown term y. Unlike in nominal unification, transforming a freshness constraint with this rule may produce new equality constraints to solve.

Rule (E5) deals with equality constraints between variables of some name sort N. We handle these constraints by noting that the way to resolve the binding scope of the names x and y is to start at the innermost binding occurrence and work towards the outside. Therefore, it should be the case that either x and y both unify with the innermost binder (x_n and y_n respectively), or that they should both be distinct from the innermost binder and unify with the next one moving outwards (i.e. x_{n-1} and y_{n-1}), and so on, or that x and y should be distinct from all of the potential binders and equal to each other. This method of dealing with equality constraints between bound names seems more natural than existing methods based on name-swapping.

The final two rules, (E7) and (E8), eliminate variables from the problem by substituting throughout the remaining constraints. They make use of a notion of substitution $\overline{c}[t/x]$ which replaces all occurrences of the variable x in $\overline{c}$ by the term t. These substitutions are capturing with respect to the abstraction term-former. Rule (E7) is the standard variable elimination rule from first-order (syntactic) unification. The side-condition $x \notin vars(t)$ on this rule enforces the *occurs check* which is necessary to avoid cyclic substitutions. The side-condition $x \in vars(\overline{c})$ ensures that this rule can only be invoked once per variable, which is necessary for termination.

Rule (E8) deals with equality constraints of the form «$x_{1..n}$»x = «$y_{1..n}$»t, where $n > 0$, the occurs check succeeds and t is not a variable, i.e. t is some compound term. We cannot simply substitute t for x here because of the preceding abstractions: x might need to be instantiated with a term syntactically different from t. For example, to satisfy the constraints $(x \# y)$ & (<x>x^* = <y>$K\,y$) it is clear that x^* must be mapped to $K\,x$, not $K\,y$. This is where swappings are necessary in nominal (and indeed equivariant) unification: however, this is not an elegant solution when bound names are represented using variables, because the potential for aliasing means that the result of a "variable swapping" $(x\,y) \cdot z$ is not unique.

Since we cannot make progress using a swapping, we note that the side-condition that t may not be a variable means that we know the outermost constructor of t. This allows us to impose some structure on the unknown term represented by x by narrowing (Antoy et al., 2000). The rules from Figure 6.2 define a narrowing relation which factors out this common functionality at unit, tuple, data and abstraction types.

The intuitive reading of $[\Delta; t] \Longmapsto [\Delta'; p]$ is that the term p represents a pattern for terms with the same outermost constructor as t. The subterms of p are variables which stand for the (as-yet unknown) subterms of the term referred to by the variable x. The new type environment Δ' is needed to ensure that the variables used to create p do not appear elsewhere in the constraint problem. They must also be mutually distinct.

Lemma 6.1.1. *If* $[\Delta; t] \Longmapsto [\Delta'; p]$ *then* $dom(\Delta) \cap dom(\Delta') = \emptyset$. □

The narrowing process is lazy in the sense that each narrowing step using rule (E8) does not replicate the entire structure of the term t but just its outermost constructor. If there is

$$\text{(N1)}\ \frac{}{[\Delta;()] \Longmapsto [\emptyset;()]} \qquad \text{(N2)}\ \frac{x \notin dom(\Delta) \qquad (K\!:\!E \to D) \in \Sigma}{[\Delta; K\,t] \Longmapsto [\{x\!:\!E\}; K\,x]}$$

$$\text{(N3)}\ \frac{\Delta \vdash (t_1, \ldots, t_m) : E_1 * \cdots * E_m \qquad x_1 \neq \cdots \neq x_m \notin dom(\Delta)}{[\Delta; (t_1, \ldots, t_m)] \Longmapsto [\{x_1\!:\!E_1, \ldots, x_m\!:\!E_m\}; (x_1, \ldots, x_m)]}$$

$$\text{(N4)}\ \frac{\Delta \vdash \texttt{<}x\texttt{>}t : \texttt{[}N\texttt{]}\,E \qquad x' \neq x'' \notin (dom(\Delta) \cup \{x\})}{[\Delta; \texttt{<}x\texttt{>}t] \Longmapsto [\{x'\!:\!N, x''\!:\!E\}; \texttt{<}x'\texttt{>}x'']}$$

Figure 6.2: Narrowing rules

more structure within the term, rule (E8) may need to be applied repeatedly. There is no rule for narrowing against variables because they have no internal structure to copy. Constraints of the form $«x_{1..n}»x = «y_{1..n}»y$ are simply left alone when $\Delta(x)$ is not a name sort—this is in direct contrast to nominal unification. It is not immediately obvious that this is correct, and we address this point in the proof of Lemma 6.3.5 below.

We conclude this section with the straightforward result that well-formedness of constraint problems is preserved by the transformation rules.

Lemma 6.1.2. *If* $\exists\Delta(\overline{c}) \longrightarrow \exists\Delta'(\overline{c}')$ *then* $\Delta' \supseteq \Delta$.

Proof. By inspection of the transition rules—(E8) is the only one which modifies Δ, clearly by expanding it. ☐

Lemma 6.1.3 (Preservation of well-formedness). *If* $\emptyset \vdash \exists\Delta(\overline{c})$ *ok and* $\exists\Delta(\overline{c}) \longrightarrow \exists\Delta'(\overline{c}')$ *then* $\emptyset \vdash \exists\Delta'(\overline{c}')$ *ok.*

Proof. By cases on the transformation rule used to derive $\exists\Delta(\overline{c}) \longrightarrow \exists\Delta'(\overline{c}')$. The proofs for the rules (F1)–(F4), (E1)–(E4), (F6) and (E6) are trivial. The cases for (F5) and (E5) require the typing side-conditions on the possible transitions. Finally, the cases for rules (E7) and (E8) require the substitutivity property for the well-formedness judgements, and case (E8) further requires the weakening property for typing judgements and the (easily verified) fact that if $\Delta \vdash t : E$ and $[\Delta; t] \Longmapsto [\Delta'; p]$ then $\Delta' \vdash p : E$. ☐

6.2 Soundness and completeness of transformations

We can now prove soundness and completeness results for the individual constraint transformation rules from Figure 6.1. We begin with a lemma which relates substitution and constraint satisfaction. This will be needed for the cases for rules (E7) and (E8) which involve substitution.

Lemma 6.2.1 (Substitution property of satisfaction). *Suppose that* $\emptyset \vdash \exists\Delta, x : E(\overline{c})$ *ok,* $\Delta \vdash t : E$, $V \in \alpha\text{-}Tree_\Sigma(\Delta, x : E)$ *and* $V(x) = [\![t]\!]_V$. *Then* $V \models \overline{c}[t/x]$ *iff* $V \models \overline{c}$. ☐

We now prove that the $\longrightarrow$ transformation rules are *sound*, i.e. that the transformation steps do not introduce any additional satisfying valuations to the problem.

Theorem 6.2.2 (Soundness of transformations). *Suppose that* $\emptyset \vdash \exists\Delta(\overline{c})$ *ok,* $\exists\Delta(\overline{c}) \longrightarrow \exists\Delta'(\overline{c}')$ *and* $V' \models \overline{c}'$ *all hold, where* $V' \in \alpha\text{-}Tree_\Sigma(\Delta')$. *Then* $V \models \overline{c}$ *holds, where* V *is the restriction of* V' *to* $dom(\Delta)$.

Proof. By case analysis on the transformation rule used to derive $\exists\Delta(\overline{c}) \longrightarrow \exists\Delta'(\overline{c}')$. The cases for rules (F1)–(F5) and (E1)–(E5) are straightforward, using standard facts about the definition of constraint satisfaction. The case for (E6) follows because $V \models x = x$ holds for any V and x. Similarly, the case for (F6) follows because $V \models x \# y$ holds for any V, x and y if $\Delta(x)$ and $\Delta(y)$ are different name sorts. The remaining cases are dealt with below.

(E7). In this case we know that $\overline{c} = (x = t) \mathbin{\&} \overline{c}^*$ and $\overline{c}' = (x = t) \mathbin{\&} \overline{c}^*[t/x]$, where $x \notin vars(t)$ and $x \in vars(\overline{c}^*)$. We also know that $\Delta' = \Delta$. By assumption we have

$$V' \models x = t \tag{6.1}$$

$$V' \models \overline{c}^*[t/x] \tag{6.2}$$

where $V' \in \alpha\text{-}Tree_\Sigma(\Delta)$. From (6.1) we get that $V'(x) = [\![t]\!]_{V'}$, and then by Lemma 6.2.1 and Lemma 6.2 we get that $V' \models \overline{c}^*$ holds, so we have $V' \models \overline{c}'$, as required.

(E8). In this case we have $\overline{c} = (\text{«}x_{1..n}\text{»}x = \text{«}y_{1..n}\text{»}t) \mathbin{\&} \overline{c}^*$ and furthermore that $\overline{c}' = (x = p) \mathbin{\&} (\text{«}x_{1..n}\text{»}p = \text{«}y_{1..n}\text{»}t) \mathbin{\&} \overline{c}[p/x]$, where t is not a variable, $x \notin vars(t)$, $n > 0$ and $[\Delta; t] \Longmapsto [\Delta^*; p]$. Furthermore, $\Delta' = \Delta, \Delta^*$. By assumption we know that

$$V' \models x = p \tag{6.3}$$

$$V' \models \text{«}x_{1..n}\text{»}p = \text{«}y_{1..n}\text{»}t \tag{6.4}$$

$$V' \models \overline{c}^*[p/x] \tag{6.5}$$

all hold, for some $V' \in \alpha\text{-}Tree_\Sigma(\Delta, \Delta^*)$. From (6.3) we know that $V'(x) = [\![p]\!]_{V'}$, (since $x \notin vars(t)$) and then by (6.5) and Lemma 6.2.1 we get that $V' \models \overline{c}^*$ holds. Furthermore, we know that $\text{«}x_{1..n}\text{»}p = \text{«}y_{1..n}\text{»}t$ is $(\text{«}x_{1..n}\text{»}x = \text{«}y_{1..n}\text{»}t)[p/x]$, because $x \notin vars(t)$. Therefore, by (6.4) and Lemma 6.2.1 we can show that $V' \models \text{«}x_{1..n}\text{»}x = \text{«}y_{1..n}\text{»}t$ holds. Thus we get that $V' \models \overline{c}'$ holds, as required.

This concludes the proof of Theorem 6.2.2. □

Next we prove that the constraint transformation rules are *complete*, i.e. that every satisfying valuation is preserved across some $\longrightarrow$ transformation of a constraint problem.

Definition 6.2.3 (Successor sets). We write $succ(\exists\Delta(\overline{c}))$ for the *successor set* of $\exists\Delta(\overline{c})$, which we define as the set $\{\exists\Delta'(\overline{c}') \mid \exists\Delta(\overline{c}) \longrightarrow \exists\Delta'(\overline{c}')\}$. ◇

Theorem 6.2.4 (Completeness of transformations). *Suppose that $\emptyset \vdash \exists\Delta(\overline{c})$ ok, $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ and $V \models \overline{c}$ all hold, and that $succ(\exists\Delta(\overline{c})) \neq \emptyset$. Then there exists $\exists\Delta'(\overline{c}') \in succ(\overline{c})$ and $V' \in \alpha\text{-}Tree_\Sigma(\Delta')$ such that $V' \models \overline{c}'$, where V and V' agree on $dom(\Delta)$.*

Proof. Since $succ(\exists\Delta(\overline{c})) \neq \emptyset$ it follows that $\overline{c}$ matches the left-hand side of one of the constraint transformation rules from Figure 6.1 and satisfies any side-conditions. Then, the proof is by case analysis on $\overline{c}$. The cases of $\overline{c}$ which match rules (F1)–(F5) and (E1)–(E5) are straightforward and follow from standard properties of constraint satisfaction.

If $\overline{c}$ is $(x = x) \mathbin{\&} \overline{c}$ the transition uses rule (E6) and the result is trivial by assumption, the problem on the right-hand side being a subset of the problem on the left-hand side. If $\overline{c}$ is $(x \# y) \mathbin{\&} \overline{c}$, where x and y are distinct variables of different name sorts, then (F6) applies and again the result follows trivially. The remaining cases are less straightforward, and we give details for these below.

Case $\overline{c} = (x = t)$ & $\overline{c}^*$. We assume that $x \notin vars(t)$ and $x \in vars(\overline{c})$ also hold. By matching against rule (E7) we get that $\Delta' = \Delta$ and $\overline{c}' = (x = t)$ & $\overline{c}^*[t/x]$. We assume that $V \models x = t$, which means that $V(x) = [\![t]\!]_V$, and we also know that $V \models \overline{c}^*$ holds. Then, we can use Lemma 6.2.1 to show that $V \models \overline{c}^*[t/x]$ holds, which gives us that $V \models \overline{c}'$, as required.

Case $\overline{c} = («x_{1..n}»x = «y_{1..n}»t)$ & $\overline{c}^*$. We also assume that t is not a variable, $x \notin vars(t)$, $n > 0$ and $[\Delta; t] \Longmapsto [\Delta''; p]$ holds for some Δ'' and p. Then, we match against rule (E8) and get that $\Delta' = \Delta, \Delta''$ and $\overline{c}' = (x = p)$ & $(«x_{1..n}»p = «y_{1..n}»t)$ & $\overline{c}^*[p/x]$. From the narrowing judgement and Lemma 6.1.1 we know that $dom(\Delta) \cap dom(\Delta'') = \emptyset$, and it follows that there exists a valuation V' which extends V from $dom(\Delta)$ to $dom(\Delta')$ such that $V' \models x = p$, i.e. such that $V'(x) = [\![p]\!]_{V'}$. Using Lemma 6.2.1 we get that $V' \models \overline{c}^*[t/x]$ holds, By the same argument we can show that $V' \models («x_{1..n}»x = «y_{1..n}»t)[t/x]$. This is equivalent to $V' \models «x_{1..n}»p = «y_{1..n}»t$ because $x \notin vars(t)$ and x cannot be one of the abstracted variables (the side-conditions enforce that $\Delta(x)$ is not a name sort). Therefore we have shown that $V' \models \overline{c}'$, as required.

This completes the proof of Theorem 6.2.4. □

The following corollary follows immediately from Theorem 6.2.2 and Theorem 6.2.4, and summarises the results of this section.

Corollary 6.2.5 (Soundness and completeness of transformations). *Suppose that $\emptyset \vdash \exists\Delta(\overline{c})$ ok and furthermore that $succ(\exists\Delta(\overline{c})) = \{\exists\Delta, \Delta_1(\overline{c}_1), \ldots, \exists\Delta, \Delta_n(\overline{c}_n)\}$. Then, for any $V \in \alpha\text{-}Tree_\Sigma(\Delta)$, $V \models \overline{c}$ iff there exists a valuation V' which extends V to $dom(\Delta, \Delta_i)$ and is such that $V' \models \overline{c}_i$, for some $i \in \{1, \ldots, n\}$.* □

6.3 Towards a decision procedure

We now use the results of the previous section to work towards a decision procedure for NonPermSat, using the constraint transformation rules presented in Figure 6.1. We begin by introducing some terminology.

Definition 6.3.1. We say that a constraint problem $\exists\Delta(\overline{c})$ is

- *terminal* (written $\exists\Delta(\overline{c}) \not\longrightarrow$) if there does not exist a constraint problem $\exists\Delta'(\overline{c}')$ such that $\exists\Delta(\overline{c}) \longrightarrow \exists\Delta'(\overline{c}')$.
- a $\longrightarrow$*-normal form* of $\exists\Delta^*(\overline{c}^*)$ if there exists a finite transformation sequence from $\exists\Delta^*(\overline{c}^*)$ to $\exists\Delta(\overline{c})$ and $\exists\Delta(\overline{c})$ is terminal.
- *strongly normalising* if all transformation sequences starting from $\exists\Delta(\overline{c})$ eventually reach a terminal constraint problem.
- *may-divergent* if there exists an infinite transformation sequence starting from $\exists\Delta(\overline{c})$. ◇

A key feature of any decision procedure is that it must always terminate, but here we run into a problem: not all constraint problems $\exists\Delta(\overline{c})$ are strongly normalising. For example, given a standard datatype `nat` for natural numbers, if $n > 0$ then the constraint problem

$$\exists\Delta((«x_{1..n}»x = «y_{1..n}»\mathrm{S}\,y) \;\&\; («y_{1..n}»y = «x_{1..n}»\mathrm{S}\,x)) \tag{6.6}$$

can be reduced to

$$\exists\Delta, x'\!:\!\mathtt{nat}, y'\!:\!\mathtt{nat}((x = \mathrm{S}\,x') \;\&\; (y = \mathrm{S}\,y') \;\&\; («x_{1..n}»x' = «y_{1..n}»\mathrm{S}\,y') \;\&\; («y_{1..n}»y' = «x_{1..n}»\mathrm{S}\,x')).$$

This is clearly may-divergent since we have recovered a variant of the original problem.

We believe that this issue can be handled using a static check which will allow us to ignore may-divergent constraint problems for the purposes of deciding satisfiability. This seems reasonable because the possibility of non-termination arises only from rule (E8) and therefore may be contained. This idea is summarised by the following conjecture.

Conjecture 6.3.2 (Decidable termination check). *There exists a decidable predicate P on constraint problems which has the following two properties:*

1. *if $\exists\Delta(\overline{c})$ is may-divergent then $P(\exists\Delta(\overline{c}))$ holds.*

2. *if $P(\exists\Delta(\overline{c}))$ holds then $\exists\Delta(\overline{c})$ is unsatisfiable.* ◇

Property 1 of Conjecture 6.3.2 suggests that if a constraint problem $\exists\Delta(\overline{c})$ is may-divergent then we can deduce this by computing whether $P(\exists\Delta(\overline{c}))$ holds. Together with property 2, this would imply that every may-divergent constraint problem is unsatisfiable. Thus, if Conjecture 6.3.2 were true then we could use P to detect may-divergent constraint problems and dismiss them as unsatisfiable. We would reserve the constraint transformation algorithm for problems which are guaranteed to be strongly normalising. This approach seems to be related to work on proving (non-)termination of term rewriting systems (Arts and Giesl, 2000; Payet, 2007).

As yet we do not have an example of a termination checking procedure P which satisfies Conjecture 6.3.2. Ideally such a procedure would not involve permutations or swappings, which are required in the equivariant unification algorithm (Cheney, 2005a). In Section 7.3 we describe the termination checking procedure for constraint problems that was used in the implementation of αML, but we do not have a proof of its correctness.

We proceed by relating the syntactic forms of constraint problems in $\longrightarrow$-normal form to their satisfiability.

Definition 6.3.3 (Solved constraint problems). A constraint problem $\exists\Delta(\overline{c})$ is *solved* iff all constraints in $\overline{c}$ have one of the following forms.

1. $x \# y$, where x and y are distinct variables and either $\Delta(x) = \Delta(y)$ or $\Delta(y)$ is not a name sort;

2. $x = t$, where $x \notin vars(t)$ and x does not appear elsewhere in $\overline{c}$;

3. $«x_{1..n}»x = «y_{1..n}»y$, where $n > 0$ and $\Delta(x)(= \Delta(y))$ is not a name sort and x and y are distinct variables; or

4. $«x_{1..n}»x = «y_{1..n}»x$, where $n > 0$ and $\Delta(x)$ is not a name sort. ◇

Lemma 6.3.4. *Any* solved *constraint problem $\exists\Delta(\overline{c})$ is also* terminal.

Proof. By cases on the possible constraints that may appear within a solved constraint problem, according to Definition 6.3.3.

1. This could match against rule (F5), but fails the side-condition as there are no preceding abstractions. It could also match against rule (F6), but cannot either because $\Delta(x)$ and $\Delta(y)$ are assumed to be the same name sort or because $\Delta(y)$ is not a name sort.

2. This could match against rule (E7) but because x does not appear elsewhere in the constraint problem the $x \in vars(\overline{c})$ side-condition fails.

3. This could match rule (E8) but cannot because the term y is a variable.

4. This could also match rule (E8) but fails for the same reason as in the previous case.

This covers all cases and completes the proof of Lemma 6.3.4. □

The relationship between terminal and solved constraints and their satisfiability is crucial to the correctness of the constraint solver. We now show that once a problem has been reduced as far as possible using $\longrightarrow$ we can determine whether it is satisfiable by examining its syntax.

Lemma 6.3.5 (Terminal constraints and satisfiability). *Let* $\exists\Delta(\overline{c})$ *be a* terminal *constraint problem such that* $\emptyset \vdash \exists\Delta(\overline{c})$ *ok. Then* $\exists\Delta(\overline{c})$ *is satisfiable iff it is* solved.

Proof. We assume that $\emptyset \vdash \exists\Delta(\overline{c})$ *ok*. By inspection of the constraint transformation rules, the possible forms of constraint in a terminal constraint problem consist of the possibilities presented in Definition 6.3.3 as well as the following:

5. $x \,\#\, x$.

6. $«x_{1..n}»K\,t = «y_{1..n}»K'\,t'$, where $K \neq K'$.

7. $x = t$, where $x \in \mathit{vars}(t)$ and t is not x.

8. $«x_{1..n}»x = «y_{1..n}»t$, where $n > 0$ and $x \in \mathit{vars}(t)$.

In particular, an equality constraint between two terms which have different numbers of outermost nested abstractions is not terminal, as it can be reduced by narrowing using rule (E8). It suffices to show that any single constraint conforming to possibilities 5–8 is unsatisfiable, and that any solved constraint problem is satisfiable. We prove these below.

Any constraint of the form 5–8 is unsatisfiable. Constraints of form 5 are not satisfiable because a name cannot be fresh for itself, and constraints of form 6 are not satisfiable because the constructors do not match. Finally, constraints of forms 7 and 8 are not satisfiable because the occurs check fails.

Any solved constraint problem is satisfiable. For a solved constraint problem $\exists\Delta(\overline{c})$ we will construct a satisfying valuation V. We write $\overline{c}_i$ for the partition of $\overline{c}$ where the constraints are all of the form $i \in \{1, 2, 3, 4\}$.

We observe that we can form a satisfying valuation $V_{3,4}$ for $\overline{c}_3 \uplus \overline{c}_4$ because the variables x and y in constraints of form 3 and x in form 4 cannot be of name sort and hence cannot coincide with any of the abstracted variables. Therefore we can simply instantiate the abstracted variables distinctly (i.e. avoiding aliasing) and instantiate the variables within the nesting to avoid the abstracted variables and satisfy the appropriate constraints.

Now we note that if $(x \,\#\, y) \in \overline{c}_1$ and $x, y \in \mathit{vars}(\overline{c}_3 \uplus \overline{c}_4)$ then $V_{3,4} \models x \,\#\, y$ by construction. Therefore we can extend $V_{3,4}$ with additional mappings which ensure that all freshnesss in $\overline{c}_1$ are satisfied, to produce a valuation $V_{1,3,4}$ which satisfies $\overline{c}_1 \uplus \overline{c}_3 \uplus \overline{c}_4$.

Finally it is always possible to extend $V_{1,3,4}$ to a satisfying valuation V for the entire problem $\overline{c}$. We begin by providing an arbitrary instantiation for any variable $z \in \mathit{vars}(\overline{c}_2)$ which only appears on the right-hand side of constraints in $\overline{c}_2$ and which has not already been instantiated. This just leaves the variables x which appear on the left-hand side of the constraints in $\overline{c}_2$. By assumption on solved constraints these variables cannot appear elsewhere in $\overline{c}$ and hence cannot have been instantiated already. Hence we are free to choose instantiations for these variables which satisfy $\overline{c}_2$.

Thus, by construction we have that $V \models \overline{c}$, as required.

We have shown that a terminal constraint problem is satisfiable precisely when it only contains constraints of the forms from Definition 6.3.3, which completes the proof of Lemma 6.3.5. □

With these results under our belt we can begin to examine the correctness of the constraint transformation algorithm. We begin by proving a soundness result: if a constraint problem has a solved $\longrightarrow$-normal form then it is satisfiable.

Theorem 6.3.6 (Soundness of algorithm). *For any constraint problem $\exists\Delta(\overline{c})$ where $\emptyset \vdash \exists\Delta(\overline{c})$ ok holds, if there exists a $\longrightarrow$-normal form $\exists\Delta'(\overline{c}')$ of $\exists\Delta(\overline{c})$ which is* solved *then $\exists\Delta(\overline{c})$ is satisfiable.*

Proof. Suppose that $\exists\Delta'(\overline{c}')$ is a $\longrightarrow$-normal form of $\exists\Delta(\overline{c})$. If $\exists\Delta'(\overline{c}')$ is solved then $\exists\Delta'(\overline{c}')$ is satisfiable by Lemma 6.3.5, i.e. there exists some $V' \in \alpha\text{-}Tree_\Sigma(\Delta')$ such that $V' \models \overline{c}'$. Finally, by Theorem 6.2.2 we get that $V \models \overline{c}$ holds (where V is the restriction of V' to $dom(\Delta)$) and hence that $\exists\Delta(\overline{c})$ is satisfiable, as required. □

We now prove a partial completeness result which is *not quite* the converse of Theorem 6.3.6 because it only applies to strongly normalising constraint problems.

Theorem 6.3.7 (Partial completeness of algorithm). *Let $\exists\Delta(\overline{c})$ be a constraint problem which is strongly normalising and is such that $\emptyset \vdash \exists\Delta(\overline{c})$ ok holds. If $\exists\Delta(\overline{c})$ is satisfiable then there exists a $\longrightarrow$-normal form $\exists\Delta'(\overline{c}')$ of $\exists\Delta(\overline{c})$ which is* solved.

Proof. If $\exists\Delta(\overline{c})$ is satisfiable then there exists a valuation $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ such that $V \models \overline{c}$ holds. Since $\exists\Delta(\overline{c})$ is strongly normalising we know that every transformation sequence eventually terminates. Then, by Theorem 6.2.4 we know that there is some sequence of transformations from $\exists\Delta(\overline{c})$ which terminate at a problem $\exists\Delta'(\overline{c}')$ such that $V' \models \overline{c}'$ holds, where V' extends V to $dom(\Delta')$. Finally, by Lemma 6.3.5 it follows that the $\longrightarrow$-normal form $\exists\Delta'(\overline{c}')$ is solved, as required. □

The assumption that $\exists\Delta(\overline{c})$ is strongly normalising is needed to ensure that it has a $\longrightarrow$-normal form which is satisfied by V. Due to these issues with termination, Theorem 6.3.6 and Theorem 6.3.7 do not constitute a decision procedure for NonPermSat. They do, however, provide a semi-decision procedure—one which never identifies an unsatisfiable constraint problem as satisfiable, but which may fail to identify some satisfiable problems (if they are may-divergent). For example, the constraint problem (6.6) would cause the constraint solver to loop but would not be wrongly identified as satisfiable.

Our experience suggests that this algorithm, while only a semi-decision procedure, is still useful in practice. Many non-terminating constraint problems are somewhat artificial and seem not to arise in real-world programs. Furthermore, we believe that this algorithm can be extended to produce a provably correct decision procedure.

Conjecture 6.3.8 (A decision procedure). *There exists a correct decision procedure for NonPermSat based on the constraint transformation rules from Figure 6.1.* ◇

In order to justify Conjecture 6.3.8 we assume the existence of a termination check with the properties described in Conjecture 6.3.2. Then we can dismiss may-divergent constraint problems as unsatisfiable without having to rewrite them using the rules from Figure 6.1. This allows us to restrict our attention to strongly normalising constraint problems $\exists\Delta(\overline{c})$, for which we can compute the finite set

$$\mathcal{S} = \{\exists\Delta'(\overline{c}') \mid \exists\Delta(\overline{c}) \longrightarrow \cdots \longrightarrow \exists\Delta'(\overline{c}') \not\longrightarrow\}$$

of $\longrightarrow$-normal forms in finite time. By Theorem 6.3.6 and Theorem 6.3.7, the constraint problem $\exists\Delta(\overline{c})$ is satisfiable precisely when there exists a solved constraint problem in $\mathcal{S}$. This is a decidable property of the syntax of $\mathcal{S}$. Therefore a proof of Conjecture 6.3.2 would give us a correct decision procedure for NonPermSat.

6.4 A tractable subproblem

In Section 3.4.1 we proved that deciding satisfiability of non-permutative constraint problems (NonPermSat) is NP-complete. We will conclude our discussion of constraint solving by isolating a tractable subproblem.

Definition 6.4.1 (Permutative problems). We say that a constraint problem $\exists\Delta(\overline{c})$ is *permutative* iff every valuation $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ such that $V \models \overline{c}$ also has $V \models \#_{nvars(\Delta)}$, where $nvars(\Delta)$ is the set of all variables x in $dom(\Delta)$ such that $\Delta(x)$ is a name sort. ◇

Intuitively, if a constraint problem is permutative then no aliasing is permitted between variables of name sort. The variables range permutatively over the underlying set of names, like the permutative names used in nominal unification (Urban et al., 2004). This suggests the following result.

Theorem 6.4.2. *PermSat is decidable in polynomial time, where PermSat is the restriction of NonPermSat to* permutative *constraint problems.* ◇

The rest of this section is devoted to a proof of Theorem 6.4.2. We define a translation of permutative constraint problems into nominal unification problems and then prove that the translation preserves satisfiability. The grammar of nominal terms used in nominal unification (Urban et al., 2004, Definition 2.3) is as follows.

$$u \quad ::= \quad n \mid \pi\, X \mid \texttt{<}n\texttt{>}u \mid K\, u \mid (u_1, \ldots, u_n) \mid ().$$

For simplicity we use the same countably infinite set *Name* of names n, which take the place of the "atoms" of (Urban et al., 2004), and which follow the permutative convention. Unification variables X are drawn from a countably infinite set *Uvar*, and a permutation π is a finite list of name-swappings (we write ι for the identity permutation). Since names and data in non-permutative constraints are both represented by the same syntactic class of variables $x \in Var$, we assume fixed bijections $\mathcal{N}$ from *Var* to *Name* and $\mathcal{U}$ from *Var* to *Uvar*.

Definition 6.4.3 (Translation of problems). We now define a translation from non-permutative terms t (as defined in (6.1)) to nominal unification terms u. The translation function is parameterised by a type environment, which we use to decide whether a particular variable x represents a name or a data term.

$$\|x\|_\Delta \triangleq \begin{cases} \mathcal{N}(x) & \text{if } \Delta(x) \text{ is a name sort.} \\ \iota\,(\mathcal{U}(x)) & \text{otherwise.} \end{cases} \qquad \|\texttt{<}x\texttt{>}t\|_\Delta \triangleq \texttt{<}\mathcal{N}(x)\texttt{>}\|t\|_\Delta \qquad \|K\,t\|_\Delta \triangleq K\,\|t\|_\Delta$$

$$\|(t_1, \ldots, t_n)\|_\Delta \triangleq (\|t_1\|_\Delta, \ldots, \|t_n\|_\Delta) \qquad\qquad \|()\|_\Delta \triangleq ()$$

We extend the translation function to non-permutative constraints, in the obvious way. To distinguish the source and target constraint languages we use the syntax of (Urban et al., 2004) for equality ($u \approx ?\, u'$) and freshness ($n \,\#?\, u'$) constraints in the world of nominal unification.

$$\begin{aligned} \|\text{«}x_{1..n}\text{»}t = \text{«}y_{1..n}\text{»}t'\|_\Delta &\triangleq \|\texttt{<}x_1\texttt{>}\cdots\texttt{<}x_n\texttt{>}t\|_\Delta \approx ?\, \|\texttt{<}y_1\texttt{>}\cdots\texttt{<}y_n\texttt{>}t'\|_\Delta \\ \|x \,\#\, \text{«}y_{1..n}\text{»}t'\|_\Delta &\triangleq \mathcal{N}(x) \,\#?\, \|\text{«}y_{1..n}\text{»}t'\|_\Delta \end{aligned}$$

Now, the representation of a non-permutative constraint problem $\exists\Delta(\overline{c})$ as a nominal unification problem is as follows.

$$\|\exists\Delta(\overline{c})\| \triangleq \{\|c\|_\Delta \mid c \in \overline{c}\}$$ ◇

Remark 6.4.4 (Types of unification variables). The term language used in (Urban et al., 2004) actually only allows unification variables at datatypes or name sorts. However, this restriction is arbitrary and there is no technical reason why unification variables of other types (such as tuple or abstraction types) could not be used in nominal unification, as we do here. $\diamond$

Given a permutative constraint problem $\exists\Delta(\overline{c})$, we first translate it to the corresponding nominal unification problem $\|\exists\Delta(\overline{c})\|$. By (Urban et al., 2004, Theorem 3.7), running the nominal unification algorithm on $\|\exists\Delta(\overline{c})\|$ either

- *fails*, which means that $\|\exists\Delta(\overline{c})\|$ has no solution; or
- produces an *idempotent most general solution* (∇, σ) to $\|\exists\Delta(\overline{c})\|$. Solutions in nominal unification are defined in (Urban et al., 2004, Definition 3.1) in terms of satisfying judgements ($\emptyset \vdash \sigma(u) \approx \sigma(u')$ and $\nabla \vdash n \mathbin{\#} \sigma(u')$) of the nominal equational logic defined in Figure 2 of that paper.

We know that nominal unification can be decided in polynomial time (Calvès and Fernández, 2008). Therefore it just remains to show that the solutions computed using nominal unification corresponds to the semantics of the original permutative constraint problem. To this end we define the notion of a ground substitution and a ground solution in nominal unification. Following the nominal unification syntax we write $\sigma(u)$ for the result of applying the substitution σ to a term u.

Definition 6.4.5 (Ground substitutions). A *ground substitution* in nominal unification is a substitution σ such that $\sigma(X)$ contains no unification variables, for all $X \in dom(\sigma)$. $\diamond$

Definition 6.4.6 (Ground solutions). If (∇, σ) is the idempotent most general solution to a nominal unification problem, a *ground solution* is any ground substitution σ^* which can be expressed as $\sigma' \circ \sigma$ for some σ', and is such that $\emptyset \vdash n \mathbin{\#} \sigma^*(X)$ holds, for all $(n, X) \in \nabla$. $\diamond$

Any most general solution (∇, σ) to a nominal unification problem denotes a set of ground solutions, which by definition are all solutions of the original problem. We begin by showing that every ground solution σ to the nominal unification version of a permutative constraint problem corresponds to a satisfying valuation V_σ of the original problem.

Definition 6.4.7. Suppose that σ is a ground solution of a nominal unification problem of the form $\|\exists\Delta(\overline{c})\|$. Then, we write V_σ for the α-tree valuation $V^* \in \alpha\text{-}Tree_\Sigma(\Delta)$ such that

- for all $x \in dom(\Delta) - nvars(\Delta)$, if $\sigma(\mathcal{U}(x)) = g$ then $V^*(x) = [g]_\alpha$; and
- for all $x \in nvars(\Delta)$, $V^*(x) = \{\mathcal{N}(x)\}$. $\diamond$

For our purposes it is sufficient to consider ground solutions: if σ^* is a ground solution and $X \in dom(\sigma^*)$ then $\sigma^*(X)$ is a ground tree g in the sense of Definition 2.2.1. Another important property of the class of α-tree valuations V_σ is that they behave permutatively on variables of name sort.

Lemma 6.4.8. *If σ is a ground solution of $\|\exists\Delta(\overline{c})\|$ then $V_\sigma \models \#_{nvars(\Delta)}$ holds.*

Proof. Straightforward from the definition of V_σ and the fact that $\mathcal{N}(x)$ is a bijection. $\square$

Lemma 6.4.9. *Suppose that $\Delta \vdash t : E$ holds and that σ is a ground substitution in nominal unification. Then it is the case that $\sigma(\|t\|_\Delta) \in [\![t]\!]_{V_\sigma}$.*

Proof. The proof is by induction on the structure of t. The interesting cases are as follows.

Case $t = x$. We must perform a case split on whether $\Delta(x)$ is a name sort or not. If so, simple calculations show that $\sigma(\|t\|_\Delta) = \mathcal{N}(x)$ and by the definition of V_σ it follows that $[\![t]\!]_{V_\sigma} = \{\mathcal{N}(x)\}$, as required.

If $\Delta(x)$ is not a name sort, we get that $\sigma(\|t\|_\Delta) = \sigma(\mathcal{U}(x))$. Once again, the result follows directly from the definition of V_σ.

Case $t = \texttt{<}x\texttt{>}t'$. In this case, $\sigma(\|t\|_\Delta) = \texttt{<}\mathcal{N}(x)\texttt{>}(\sigma(\|t'\|_\Delta))$. Using the inductive hypothesis we can show that $\sigma(\|t'\|_\Delta) \in [\![V_\sigma]\!]_{t'}$. In order to derive that $\sigma(\|\texttt{<}x\texttt{>}t'\|_\Delta) \in [\![V_\sigma]\!]_{\texttt{<}x\texttt{>}t'}$ we use Lemma 6.4.8 and a result similar to Lemma 2.3.4. This shows that when the α-equivalence class $[\![\texttt{<}x\texttt{>}t']\!]_{V_\sigma}$ is formed, precisely those names corresponding to $\mathcal{N}(x)$ are captured.

The cases for unit, tuple and data terms are straightforward. □

We now show that every ground substitution in nominal unification corresponds to a satisfying valuation of the original permutative constraint problem. This proof makes heavy use of the *adequacy* result (Urban et al., 2004, Proposition 2.16), which shows that the $\nabla \vdash u \approx u'$ and $\nabla \vdash n \,\#\, u'$ judgements from that paper correspond to the usual notions of α-equivalence and "not free in" (and have $\nabla = \emptyset$) when u and u' are both ground.

Lemma 6.4.10. *Suppose that σ is a ground solution to $\|\exists\Delta(\overline{c})\|$ and that $c \in \overline{c}$. Then:*

(i) if c is $t \texttt{=} t'$ and $\emptyset \vdash \sigma(\|t\|_\Delta) \approx \sigma(\|t'\|_\Delta)$ then $V_\sigma \models t \texttt{=} t'$.

(ii) if c is $x \,\texttt{\#}\, t'$ and $\emptyset \vdash \mathcal{N}(x) \,\#\, \sigma(\|t'\|_\Delta)$ then $V_\sigma \models x \,\texttt{\#}\, t'$.

Proof. We prove the two sentences below.

(i) We assume that c is $t \texttt{=} t'$ and that $\emptyset \vdash \sigma(\|t\|_\Delta) \approx \sigma(\|t'\|_\Delta)$. Using the adequacy result (Urban et al., 2004, Proposition 2.16) this gives us that $\sigma(\|t\|_\Delta) =_\alpha \sigma(\|t'\|_\Delta)$ holds. By Lemma 6.4.9 we get that $[\![t]\!]_{V_\sigma} = [\![t']\!]_{V_\sigma}$. This is equivalent to $V_\sigma \models t \texttt{=} t'$, as required.

(ii) We assume that c is $x \,\texttt{\#}\, t'$, and also that $\emptyset \vdash \mathcal{N}(x) \,\#\, \sigma(\|t'\|_\Delta)$ holds. Using adequacy again, we get that $\mathcal{N}(x) \notin FN(\sigma(\|t'\|_\Delta))$. By Lemma 6.4.9 we get that $\sigma(\|t'\|_\Delta) \in [\![t']\!]_{V_\sigma}$, and it follows that $V_\sigma(x) \notin FN([\![t']\!]_{V_\sigma})$. Therefore we get that $V_\sigma \models x \,\texttt{\#}\, t'$ holds, as required.

This completes the proof of Lemma 6.4.10. □

We now prove the reverse direction, which is that any satisfying valuation V of a permutative constraint problem corresponds to a ground solution σ_V of the corresponding problem in nominal unification. This technical development is also closely related to that from Section 5.4.4 because the semantics of our constraint problems are phrased in terms of α-equivalence classes of ground trees. Therefore the translation from V to σ_V is up to some permutation π_V, in the following sense.

Lemma 6.4.11. *If $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ and $V \models \texttt{\#}_{nvars(\Delta)}$ then there exists a permutation π_V such that $V(x) = \{\pi_V(\mathcal{N}(x))\}$ for all $x \in nvars(\Delta)$.*

Proof. Since $V \models \texttt{\#}_{nvars(\Delta)}$ holds, we know that if x, y are distinct variables in $nvars(\Delta)$ then $V(x) \neq V(y)$. This, and the fact that $\mathcal{N}(x)$ is bijective, implies that a suitable π_V exists. □

Definition 6.4.12 (Translation of valuations). Given an α-tree valuation $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ such that $V \models \texttt{\#}_{nvars(\Delta)}$ we write σ_V for any substitution where

- $dom(\sigma_V) = \{\mathcal{U}(x) \mid x \in dom(\Delta) - nvars(\Delta)\}$; and
- $\pi_V \cdot \sigma_V(\mathcal{U}(x)) \in V(x)$ for all $x \in dom(\Delta) - nvars(\Delta)$,

where π_V is as defined in the statement of Lemma 6.4.11. ◇

We are now in a position to prove the converse of Lemma 6.4.10.

Lemma 6.4.13. *Suppose that $V \in \alpha\text{-}Tree_\Sigma(\Delta)$, where $V \models \#_{nvars(\Delta)}$, and that $\Delta \vdash t : E$. Then, we have that $\pi_V \cdot \sigma_V(\|t\|_\Delta) \in [\![t]\!]_V$.*

Proof. By induction on t. The proof is similar to the proof of Lemma 6.4.9 above and to the proof of Lemma 5.4.7 from Section 5.4.4, so we omit much of the detail here. The $V \models \#_{nvars(\Delta)}$ assumption guarantees the existence of the permutation π_V (see Lemma 6.4.11 above). The base case for variables x follows from the properties of π_V if $\Delta(x)$ is a name sort, and from the definition of σ_V otherwise. We also use Lemma 2.3.4. □

Lemma 6.4.14. *Suppose that $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ and $\Delta \vdash c$ ok. If $V \models \#_{nvars(\Delta)}$ then*

(i) if c is $t = t'$ and $V \models t = t'$ then $\emptyset \vdash \sigma_V(\|t\|_\Delta) \approx \sigma_V(\|t'\|_\Delta)$.

(ii) if c is $x \# t'$ and $V \models x \# t'$ then $\emptyset \vdash \mathcal{N}(x) \# \sigma_V(\|t'\|_\Delta)$.

Proof. We prove the two sentences separately.

(i) We suppose that c is $t = t'$, and that $V \models t = t'$ holds, i.e. that $[\![t]\!]_V = [\![t']\!]_V$. By Lemma 6.4.13 there exists a permutation π_V and a ground substitution σ_V such that $\pi_V \cdot \sigma_V(\|t\|_\Delta) \in [\![t]\!]_V$ and $\pi_V \cdot \sigma_V(\|t'\|_\Delta) \in [\![t']\!]_V$ both hold, and it follows that $\pi_V \cdot \sigma_V(\|t\|_\Delta) =_\alpha \pi_V \cdot \sigma_V(\|t'\|_\Delta)$. By equivariance, this is equivalent to $\sigma_V(\|t\|_\Delta) =_\alpha \sigma_V(\|t'\|_\Delta)$. Finally, by the adequacy result from (Urban et al., 2004, Proposition 2.16) we get that $\emptyset \vdash \sigma_V(\|t\|_\Delta) \approx \sigma_V(\|t'\|_\Delta)$ holds, as required.

(ii) If c is $x \# t'$, we assume that $V \models x \# t'$. This is equivalent to $V(x) \notin FN([\![t']\!]_V)$: therefore there exists n such that $V(x) = \{n\}$ and $n \notin FN([\![t']\!]_V)$. By Lemma 6.4.13 there exists a permutation π_V and a ground substitution σ_V such that $\pi_\pi \cdot \sigma_V(\|t'\|_\Delta) \in [\![t']\!]_V$ and $V(x) = \{\pi_V(\mathcal{N}(x))\}$. It follows that $\pi_V(\mathcal{N}(x)) = n$ and since all members of an α-equivalence class have the same free names we get that $\pi_V(\mathcal{N}(x)) \notin FN(\pi_\pi \cdot \sigma_V(\|t'\|_\Delta))$. By equivariance we get $\mathcal{N}(x) \notin FN(\sigma_V(\|t'\|_\Delta))$, and using the adequacy theorem again gives us that $\emptyset \vdash \mathcal{N}(x) \# \sigma_V(\|t'\|_\Delta)$ holds, as required.

This completes the proof of Lemma 6.4.14. □

Having shown that every satisfying valuation of a permutative constraint problem corresponds to a ground solution of the nominal unification version, and vice versa, we are now in a position to prove Theorem 6.4.2.

Proof (of Theorem 6.4.2). We can decide satisfiability of a permutative constraint problem by translating it to the corresponding nominal unification problem $\|\exists\Delta(\overline{c})\|$ and using a nominal unification algorithm which has polynomial time complexity (Calvès and Fernández, 2008). By Lemma 6.4.10 and Lemma 6.4.14, we can compute a (most general) solution in nominal unification iff there exists a satisfying α-tree valuation for the original constraint problem. Therefore, satisfiability of a permutative constraint problem can be decided in polynomial time via translation to nominal unification. □

Intuitively, this result holds because at most one of the non-determistic possibilities in rules (F5) and (E5) from Figure 6.1 can be satisfied. Finally, since Definition 6.4.1 gave a semantic property, we provide a syntactic approximation.

Definition 6.4.15 (Explicitly permutative problems). A constraint problem $\exists\Delta(\overline{c})$ is *explicitly permutative* iff $(x \# y) \in \overline{c}$ for all distinct variables $x, y \in nvars(\Delta)$. $\diamond$

It is clear if a constraint problem is explicitly permutative then it is permutative. Furthermore we can decide in polynomial time whether a constraint problem is explicitly permutative by examining its syntax.

Chapter 7

Implementation

"It's all very well in practice, but it will never work in theory."
—French proverb

As we argued in Chapter 1, a prototype implementation can be very useful during the early stages of the language design process and the design of αML was no different in this respect. We went through several prototypes before reaching the one described here, which places slightly more emphasis on efficiency. The implementation described in this chapter is available online from the author's web page.

We begin this chapter by describing the overall structure of the implementation before outlining the extended version of αML that is implemented. We then discuss our implementation of the constraint transformation algorithm described in Chapter 6. The reader is referred to Appendix C, which describes the compilation process and the αML runtime in greater detail, and to Appendix D, which presents some examples of complete αML programs and example sessions with the interpreter.

7.1 Interpreter overview

The αML implementation is structured as a toplevel interpreter: the user enters an expression into the terminal and the result of the computation (if any) is printed back to the terminal. The interpreter was written in the Objective Caml (OCaml) programming language.

Figure 7.1 gives an overview of the internal structure of the αML interpreter. User input is first lexed and parsed to produce an abstract syntax tree in an intermediate representation where variables are still strings and all expressions carry location information. This allows

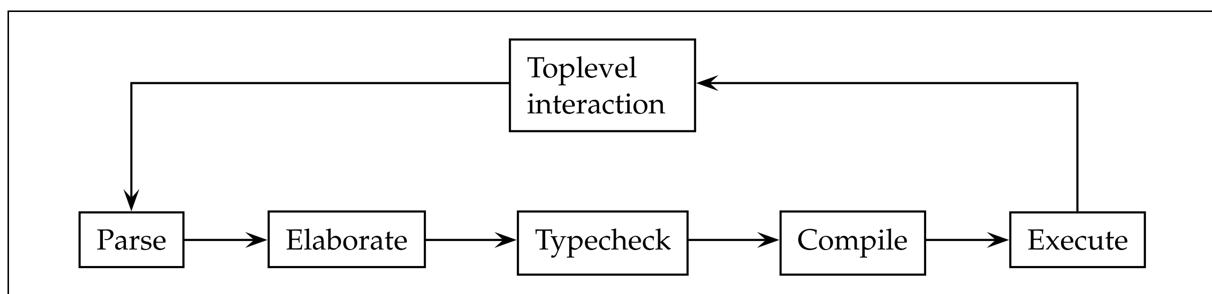


Figure 7.1: Interpreter structure

comprehensible error messages to be produced. The intermediate representation is then elaborated to remove syntactic sugar (this step is discussed further in Section 7.2 below). This results in a language which closely resembles the core αML language from Chapter 3.

Assuming that the program passes the typechecker, it can be evaluated safely (in the sense of Theorem 3.7.7 and Theorem 3.7.8). The intermediate representation is compiled into a custom bytecode for the αML bytecode machine—this process is described in Appendix C. The bytecode program is executed and any results are pretty-printed back to the user. We use the term *bytecode machine* here to avoid confusion with the *abstract machine* used in Chapter 3 to present the operational semantics of αML. Certain aspects of the implementation are of particular interest and we discuss these below.

- **Implementation of non-determinism and finite failure.**
  The practical implementation of the non-deterministic search features of αML was left unspecified in the operational semantics presented in Chapter 3. Our implementation of the αML bytecode machine maintains a collection of computations (encoded as a list of states of the bytecode machine). A scheduling function repeatedly calls a single-step transition function `exec` which takes a single state and returns a list of states, since any individual instruction could cause a branch. The scheduler then decides which state to reduce next: both depth-first and breadth-first search strategies are supported. If a particular computation fails finitely (for example, if a constraint is unsatisfiable) then the `exec` function returns an empty list.

- **Internal representation of binders.**
  αML provides support for object-language binders via nominal abstract syntax, but within the interpreter itself the meta-language binders are represented using de Bruijn indices (see Section 1.3). These are used as pointers into an environment which is just a list of the machine values referred to by bound variables. The compilation process (described in Appendix C) uses a symbol table which maps variable names to integers to ensure that the indices correctly represent the binding structure of the original program.

- **Treatment of existential variables.**
  The presentation of αML in Chapter 3 and subsequent chapters uses normal value identifiers to stand for existential variables, so evaluation can sometimes involve open code. While this is a neat theoretical trick, and produced a very elegant operational semantics, it proved inefficient and difficult to implement. Therefore the implementation treats existential variables slightly differently.

  In the interpreter, when an existential variable is generated it produces a machine value $\texttt{MEvar}(n)$ (described in Appendix C). The integer tag n records that this is the nth existential variable generated so far. Machine values representing existential variables are treated no differently from any other machine value, but they are distinct from meta-level variables. When existential variables are pretty-printed, the integer is converted to a string so that existential variables appear to the user as follows.

  $$\texttt{?a} \quad \texttt{?b} \quad \cdots \quad \texttt{?y} \quad \texttt{?z} \quad \texttt{?aa} \quad \cdots$$

  We write $\mathcal{N}(\texttt{?x})$ for the integer corresponding to `?x` and $\mathcal{N}^{-1}(n)$ for the inverse function, which computes the string representation of the nth existential variable. The interpreter also instantiates existential variables at certain points for efficiency reasons, even though this is not specified in the operational semantics.

- **Constraints and pretty-printing.**
  During evaluation, the interpreter gathers constraints which have been shown to be consis-

tent using the algorithm described in Chapter 6. When pretty-printing the results of computations, some of the implicit substitution information is used to make the answers more readable. This is achieved by recursively "walking" the value and instantiating existential variables where possible, although no freshness information is currently printed. The user experience is illustrated by example in Appendix D.

7.2 Extended αML

To simplify the development of the theory, the αML language presented in Chapter 3 was deliberately made as small as possible. We removed as many features as possible to produce a minimal calculus for animating inductive definitions. However, the lack of high-level programming abstractions means that programs can get quite verbose. The implementation of αML addresses this issue by extending the input language to provide a more usable interface to programmers. The theoretical properties of the core language are preserved because many of the new features are expressible in core αML. In this section we will describe the additions to the input language compared to core αML, and translate the new features into the core language where appropriate.

The first major change is that we have dropped the requirement that αML programs be in A-normal form (Flanagan et al., 1993). This removes the need to add extra `let` bindings to programs, making them much more concise and readable. We have also added some straightforward constructs such as wildcard variables in patterns, anonymous non-recursive functions ($\texttt{fn}\,(x\,{:}\,T) \to e$), sequential composition ($e \,\&\, e'$, as mentioned in Section 4.1) and shorthand notation for defining functions. These are all implemented by translation into core αML.

We have also implemented fresh name generation as defined in Section 5.7, along with some more complex enhancements which are summarised below. We defer discussion of further possible extensions to the language until Section 8.1.

- **Existential variable generation.**
  We actually define the $\exists x\,{:}\,E.\,e$ binding construct of core αML in terms of an expression `some` E. This simply returns a newly-generated existential variable as opposed to binding it in the expression e (generating and returning a new existential variable seems to be a commmon programming idiom). The `some` construct is implemented directly in the αML compiler, and the existential binder `exists` is defined in terms of `some`, as follows.

$$\texttt{exists}\,x\,{:}\,E.e \;\triangleq\; \texttt{let}\,x\,\texttt{=}\,(\texttt{some}\;E)\;\texttt{in}\;e.$$

- **Distinct name constraints.**
  Another common requirement is to constrain a finite set of name-variables to be distinct from one another, as defined in Definition 2.5.8. This is useful if we wish to enforce the permutative convention. We introduce a notation $\texttt{distinct(}x_1\texttt{,}\ldots\texttt{,}x_n\texttt{)}$ as a shorthand for the set of freshness constraints $\{x_i \,\#\, x_j \mid 1 \le i < j \le k\}$. It is more efficient to process the set of constraints at once than to deal with the freshnesses individually.

- **Unbinding constructs.**
  The core αML language defined in Chapter 3 does not provide convenient syntax for deconstructing abstraction values, so we add the following syntactic sugar (see Section 5.7 for a discussion of generative unbinding).

$$\begin{array}{l}\texttt{unbind}\,e\,\texttt{as}\,\texttt{<}x\texttt{>}x'\,\texttt{:}\,\texttt{[}N\texttt{]}\,E\;\texttt{in}\;e' \;\triangleq \\ \qquad \texttt{let}\,x\,\texttt{=}\,(\texttt{some}\;N)\;\texttt{in}\;\texttt{let}\,x'\,\texttt{=}\,(\texttt{some}\;E)\;\texttt{in}\;(e\,\texttt{=}\,\texttt{<}x\texttt{>}x')\,\&\,e'.\end{array}$$

The unbind operator uses existential variables to form a pattern `<x>x'`. The pattern-matching itself is done using the constraint solving algorithm described in Chapter 6. This means that the type of the abstraction body is restricted to be an equality type (reflected in the type of x'). This is strictly less general than generative unbinding in FreshML (Pitts and Shinwell, 2008), where the body of the abstraction may be of any type, including function types.

In general, when unbinding a name using unbind one must manually insert some freshness constraints to prevent name capture. For example, the capture-avoiding substitution function on λ-terms can be defined as follows

```
let rec sub (t':lam) (x:var) (t:lam) : lam = case t' of
  V y -> (x=y & t) || (x#y & V y)
| A w -> let t1 = w.1 in let t2 = w.2 in
         A((sub t1 x t),(sub t2 x t))
| L z -> unbind z as <y>t'':[var]lam in
         y#(x,t) & L<y>(sub t'' x t);;
```

where in the final clause the freshness constraint $y \mathbin{\#} (x, t)$ is the usual one for ensuring capture-avoidance. In FreshML these freshnesses are implicit in the dynamics of generative unbinding, but we argue that requiring the assertions forces the programmer to think long and hard about freshness and capture-avoidance, which is no bad thing.

The implementation also supports the unbind_fresh variant defined in (5.10), which uses fresh instead of some to generate the pattern variable x.

- **Success and finite failure.**
  The αML interpreter uses the syntax yes for the value of type prop which denotes a successful proof-search computation. Dually, there is a syntax which causes the current computation to fail immediately: no T. We define this construct for any type T because we can define a failing expression at any type (recall (5.9) from Section 5.6).

  As a motivating example, we consider the problem of defining the lookup of a variable in a typing environment as a partial function

$$\texttt{lookup} : \texttt{tenv} * \texttt{var} \longrightarrow \texttt{type}$$

  where tenv is a type representing environments Γ. To fit with the logic programming notion of failing when a result cannot be found, we specify that lookup (Γ, x) should fail finitely if $x \notin dom(\Gamma)$. Such a function could be defined in αML as follows

```
let rec lookup (gamma:tenv) (x:var) : type = case gamma of
  Nil _ -> no type
| Cons z ->  let y = z.1.1 in let t = z.1.2 in let gamma' = z.2 in
   (x=y & t) || (x#y & lookup gamma' x);;
```

  where we use the no type construct to fail finitely. Note the use of various other defined forms such as the shortcut recursive function definition and the sequential composition operator which help make the code readable.

- **Inductive definitions.**
  An expression corresponding to an inductive definition is delimited by double braces: "{{" at the beginning and "}}" at the end. Within these delimiters there must be one or more inductive definitions, each of which takes the form

$$\frac{e}{R\,(p_1\,,\,\ldots\,,p_n)}\ [\textit{rulename}\ \texttt{where}\ x_1 : E_1, \ldots, x_k : E_k]$$

R is a *relation symbol*—these are declared using the following syntax.

$$\texttt{relation}\ R_1\ \texttt{<:}\ E_1\ \texttt{and}\ \cdots\ \texttt{and}\ R_n\ \texttt{<:}\ E_n$$

When a relation symbol R_i is applied to an expression of the appropriate equality type E_i the resulting term is of a special datatype S_r, which is reserved for terms corresponding to instances of inductively defined relations. The premise of every inference rule must have type S_r. In the implementation, S_r is referred to as `rel`. The conclusion of a rule contains schematic patterns $p_1, \ldots, p_n$ (enclosed by parentheses), which follow the syntax of Definition 2.3.1.

On the right-hand side of the rule the *rulename* label is optional—it currently serves no technical purpose and is for documentation only. A type annotation $x_i : E_i$ is required for any value identifier appearing in the rule which is scoped to within that rule (it is possible to write rules with free variables, e.g. in a definition which is parameterised by some argument).

The interpreter automates the process of encoding inductive definitions in αML which was described in Chapter 4, and automatically translates the inductive definition syntax into the corresponding recursive function in core αML. The example code in Appendix D makes extensive use of this syntactic sugar.

- **Ground trees.**
  We have also automated the translation of ground trees into αML expressions as defined in Section 5.4. This syntactic form is delimited by "`[|`" and "`|]`", and the syntax of the ground trees themselves is given by the following grammar.

$$g \ ::= \ x \mid () \mid K\,g \mid (g_1, \ldots, g_n) \mid \texttt{<}x\,{:}\,N\texttt{>}g.$$

The concrete syntax is almost identical to the grammar of schematic patterns from Definition 2.3.1. The only syntactic difference is that every variable in abstraction position must have a type annotation with the sort of that particular bound name. This is because the translation uses existentially-quantified variables to represent the bound names of the ground tree, and the typechecker requires type annotations for each of these.

The crucial semantic difference between this sub-language of ground trees and the rest of the language is how value identifiers are interpreted. Between `[|` and `|]`, the permutative convention applies, so the value identifiers (which must all be of name sorts) are interpreted as standing for distinct object-language names, like the permutative names (atoms) of FreshML (Shinwell, 2005). The translation into core αML ensures that these expressions are contextually equivalent precisely when the corresponding ground trees are α-equivalent. As an example, the representation of the λ-term $\lambda x.\,\lambda y.\,(x\,y)$ is as follows.

```
[| Lam <x:var>(Lam <y:var>(App (Var x,Var y))) |]
```

The user does not have to write the freshness constraint $x \,\#\, y$—this is added automatically by the translation into core αML.

Since the free names of a ground tree g are interpreted as free value identifiers of its encoding (see Section 5.4.2) then any free names in a ground tree expression must be declared beforehand (with the right types). For example, to encode the open λ-term $\lambda x.\,(x\,y)$ the variable y must have been declared already.

```
let y = some var;;
[| Lam <x:var>(App (Var x,Var y)) |];;
```

Furthermore, the evaluation of an encoded ground tree may fail finitely because there might already be constraints between the variable corresponding to the free names of the ground tree (recall that the translation of ground trees requires that all of the names appearing in the tree be pairwise distinct). For example, the following interaction leads to finite failure.

```
let y = some var;;
let z = some var;;
y = z;; (* constrain two names to be equal *)
[| Lam<x:var>(App(Var y,Var z)) |];; (* fails finitely on y#z *)
```

7.3 Implementing the constraint solver

In this section we outline an implementation of the constraint transformation algorithm described in Chapter 6.

The constraint transformation algorithm is divided into a deterministic phase (which incorporates a termination checking procedure) and a non-deterministic phase. The deterministic phase implements all of the transformation rules from Figure 6.1 except for the branching rules (F4) and (E4) and the narrowing rule (E8). The deterministic transformations are straightforward to implement and are not discussed further here for reasons of space. We will focus on the termination checker and on the non-deterministic rules (F4) and (E4), which make the decision problem computationally expensive.

The αML implementation augments the constraint transformation algorithm described in Chapter 6 with a termination checking procedure. This re-uses the deterministic transformation rules to perform a restricted form of first-order unification on the leftover equality constraints to check for cyclic dependencies. If the termination check is passed then all of the narrowing steps using rule (E8) can be performed in one go, followed by a final application of the other deterministic transformation rules. Since the termination check has been passed, the deterministic transformation phase must have reached a normal form at this point. The termination checker has not been proved correct but seems to work well in practice.

We now turn our attention to the non-deterministic rules (F4) and (E4). Solving constraints between variables of name sort by actually substituting one variable for another would be a fairly inefficient strategy. Therefore, we check satisfiability of constraints between variables of name sort by arranging the variables into equivalence classes. Two variables are in the same equivalence class precisely when they are aliased to stand for the same concrete name. Freshness (i.e. name inequality) constraints are recorded between equivalence classes as opposed to between particular variables. This data structure is represented as a graph using the OCAMLGRAPH library (Conchon et al., 2008). Each vertex is labelled with a list of variables which represents an equivalence class (a particular variable can appear in at most one vertex label). An edge joining two vertices represents a freshness constraint between those equivalence classes, and if two vertices are not connected then the two equivalence classes might refer to two different names or they might be aliased. This data structure is a simpler variant of Cheney's *permutation graphs* (Cheney, 2005a). The two graphs in Figure 7.2 give an example of this representation of constraints (we assume that all variables here are of the same name sort).

Graph 1 from Figure 7.2 represents the constraint problem $(x = y)$ & $(x \# z)$ & $(w \# y)$, and if we add the extra constraints $(q = w)$ & $(z \# q)$ we get graph 2. The new equality constraint is represented by adding q to the vertex label that contains w and the new freshness constraint produces the extra edge in the graph. Failure is detected if we try to add an equality constraint between variables whose vertices are connected by a freshness edge (e.g. $y = z$ here) or if we try to add a freshness constraint between variables in the same vertex label (e.g. $q \# w$).

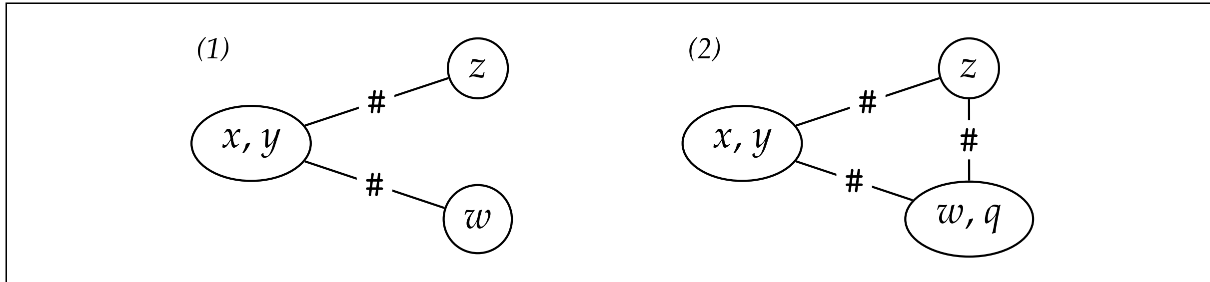


Figure 7.2: Examples of constraint representation

The non-deterministic transformation phase performs a depth-first search over the (finite) tree of possible aliasing patterns between variables of name sort. The implementation uses a wrapper around the imperative graph implementation from OCAMLGRAPH which records low-level changes to the graph and maintains a stack of inverse operations to undo these changes. This makes it easy to roll the graph back to an earlier state when backtracking.

Chapter 8

Conclusions and future work

"The Road goes ever on and on,
out from the door of where it began.
And now far ahead the Road has gone,
let others follow it who can."
—J. R. R. Tolkien

At the beginning of this dissertation I proposed the following thesis:

> *Rapid and correct prototyping of programming languages is possible using the executable meta-language αML.*

The intervening chapters have presented evidence to support this thesis. The main contributions of this dissertation are:

- the definition of a language of rule-based definitions involving binders, called α-inductive definitions, as a formal model of object-languages;
- the design of an executable meta-language, αML, for quickly and easily encoding and animating α-inductive definitions; and
- correctness proofs for αML with regard to the encodings of α-inductive definitions and the representation of binders up to α-equivalence.

These contributions support my thesis because they demonstrate that αML is, at least in principle, a suitable meta-language for rapidy and correctly prototyping programming languages and other systems defined using schematic inference rules. We say "in principle" because there is still much to be done before αML is a viable language for prototyping large-scale languages and testing those prototypes on non-trivial examples. There are issues relating to efficiency and to ease of use, which we will discuss in the next section.

From a programmer's perspective, the αML language exposes little of its theoretical basis in nominal abstract syntax: the abstraction term-former ($<x>t$) and the freshness constraint syntax ($x \# t$) are the only giveaways. We see it as beneficial that an end-user needs very little knowledge (if any) of the underlying semantics in terms of nominal sets. The same is not true of many similar systems, in particular those based on higher-order representation techniques. Information hiding is a key theme throughout computer science and we believe that this is an important step: for nominal methods to gain widespread use we must hide as much internal detail as possible.

An interesting and unexpected by-product of this work has been the study of the constraint problem NonPermSat and the associated constraint transformation algorithm. Syntactically

speaking, our constraint problem is a subproblem of the "*algorithmically involved*" equivariant unification problem (Cheney, 2005a). However, our the proof that NonPermSat is NP-complete (Section 3.4.1) demonstrates that it is in fact equivalent to equivariant unification. This fact might allow us to reuse the well-developed theory of $\sim$-resolution (Cheney and Urban, 2008) but implement it in terms of our constraint problem.

8.1 Future work

It would be particularly useful to formalise the language and mechanically verify the proofs of correctness of αML, such as those from Chapter 4 and Chapter 5. It might also be possible to formalise the compilation process and runtime described in Appendix C, in order to show that the compilation function preserves the semantics of programs. In the rest of this section we outline some other possible directions for future work related to αML.

8.1.1 Development of theory

Expressiveness of α-inductive definitions

To our knowledge there are no concrete results in the literature which relate the expressiveness of the various formalisations of the notion of inductive definitions. Translations exist between constraint problems such as nominal and higher-order pattern unification (see Section 8.1.5 below for a brief discussion) but not for the sets of relations which are definable using the various encodings of abstract syntax with binders mentioned in Chapter 1.

There are concrete mathematical questions to be answered here: for example, let us assume a formal notion of inductively-defined relations in some other encoding, for example weak HOAS (Despeyroux et al., 1995), along with a translation between (α-equivalence classes of) ground trees and weak HOAS terms. Then we might ask: for every α-inductive definition $\mathcal{D}$, does there exist a weak HOAS definition which carves out precisely the same set of ground terms (and vice versa)?

Results such as this could bring some mathematical rigour to the debates between various encodings of abstract syntax. However, just because an equivalent encoding exists does not mean that that encoding is easy to construct, especially for a non-expert: there will always be a pragmatic aspect to this issue.

Fresh name generation

As mentioned in Section 5.7, it is not known whether the `fresh` operator for generating globally fresh names is definable in the core αML language defined in Chapter 3. It would be very interesting to settle this question by finding a proof of, or a counter-example to, Conjecture 5.7.1.

Although it seems unlikely that the `fresh` construct itself can be defined directly in terms of core αML, we believe that programs in a language which uses `fresh` can be translated into αML via a whole-program transformation. Such a translation could use a monadic programming style to pass the list of generated variables around the program as explicit state so that the fresh name generation operation can be implemented using the operations available in αML.

To illustrate this we briefly describe a translation of Mini-FreshML (Shinwell, 2005, Figure 3.1) into core αML. For simplicity we will assume that our datatype definition Σ contains a single name sort N and the following datatype definitions.

```
datatype bool = True of unit | False of unit
datatype env = Nil of unit | Cons of N * env
```

To save space, we will temporarily ignore the A-normal form restriction on αML programs imposed in Chapter 3, and we will abbreviate $\mathtt{Cons}\,(v_1\,,\,v_2)$ by $v_1 :: v_2$. We will also write $\lambda x\!:\!T.\,e$ for an anonymous (non-recursive) function and abuse the syntax for `let` bindings to deconstruct pairs in the obvious way.

A Mini-FreshML expression e of type T is translated to a core αML expression $\ulcorner e \urcorner$ of type $\mathtt{env} \to (\mathtt{env},\, T)$. This function threads the current environment ε of generated names through the execution. Some interesting cases in the definition of $\ulcorner e \urcorner$ are given below.

$$
\begin{array}{rcl}
\ulcorner <e_1>e_2 \urcorner & \triangleq & \lambda\varepsilon\!:\!\mathtt{env}.\ \mathtt{let}\ (\varepsilon_1\,,\,v_1) = \ulcorner e_1 \urcorner(\varepsilon)\ \mathtt{in} \\
 & & \quad \mathtt{let}\ (\varepsilon_2\,,\,v_2) = \ulcorner e_2 \urcorner(\varepsilon_1)\ \mathtt{in}\ (\varepsilon_2\,,\,<v_1>v_2) \\
\ulcorner \mathtt{fresh} \urcorner & \triangleq & \lambda\varepsilon\!:\!\mathtt{env}.\ \exists x\!:\!N.\ x \,\#\, \varepsilon \,\&\, (x :: \varepsilon\,,\,x) \\
\ulcorner \mathtt{swap}\ e_1, e_2\ \mathtt{in}\ e_3 \urcorner & \triangleq & \lambda\varepsilon\!:\!\mathtt{env}.\ \mathtt{let}\ (\varepsilon_1\,,\,x_1) = \ulcorner e_1 \urcorner(\varepsilon)\ \mathtt{in} \\
 & & \quad \mathtt{let}\ (\varepsilon_2\,,\,x_2) = \ulcorner e_2 \urcorner(\varepsilon_1)\ \mathtt{in}\ \mathtt{let}\ (\varepsilon_3\,,\,q) = \ulcorner e_3 \urcorner(\varepsilon_2)\ \mathtt{in} \\
 & & \quad \exists w\!:\!E.\ <x_1><x_2>w = <x_2><x_1>q \,\&\, (\varepsilon_3\,,\,w) \\
\ulcorner \mathtt{let}\ <x>x' = e_1\ \mathtt{in}\ e_2 \urcorner & \triangleq & \lambda\varepsilon\!:\!\mathtt{env}.\ \mathtt{let}\ (\varepsilon_1\,,\,q) = \ulcorner e_1 \urcorner(\varepsilon)\ \mathtt{in} \\
 & & \quad \exists x\!:\!N.\ \exists x'\!:\!E.\ x \,\#\, \varepsilon_1 \,\&\, <x>x' = q \,\&\, \ulcorner e_2 \urcorner(x :: \varepsilon_1)
\end{array}
$$

The case for abstraction expressions is straightforward: the components are evaluated (with the state being threaded through) and the resulting values are put together in an αML abstraction value. In the `fresh` case we can use the normal existential variable generation operation from αML to simulate fresh name generation because we have the environment ε which holds all names generated so far. This allows us to use an explicit freshness constraint ($x \,\#\, \varepsilon$) to make the new name x behave like a globally fresh name.

The cases for swapping and abstraction deconstruction highlight a limitation of the encoding: we can only translate Mini-FreshML expressions where the bodies of the swapping and abstraction decomposition sub-expressions correspond to αML equality types. This is because we use constraint solving to simulate the swapping and deconstruction operations, which is only decidable at equality types. This limitation does not seem too serious because many Mini-FreshML programs seem to satisfy this criterion in practice.

The encoding of swapping uses a slightly odd construction with two nested abstractions to simulate the swapping operation. From the semantics of non-permutative constraints it follows that any valuation V satisfying this equality constraint will have $V(w) = (V(x_1)\,V(x_2)) \cdot V(q)$ and vice versa (since the swapping operation is self-dual). We are abusing the swapping syntax here, but hopefully the meaning is clear.

Finally, the encoding of an abstraction deconstruction expression uses a construction like the encoding of `fresh` to create a globally fresh name to stand for the bound name of the abstraction. We do not use explicit swapping but rather an equality constraint ($<x>x' = q$) to deconstruct the abstraction. Because we have asserted that x is fresh for ε_1 we know that there will be no inadvertent name capture.

We do not have a proof that this encoding is correct: this is left for future work. Ideally we would like to relate the result of evaluating a Mini-FreshML program to the result of evaluating its αML translation and also show that the translation preserves observational equivalence. The theory of observational equivalence for Mini-FreshML is well developed (Shinwell, 2005).

8.1.2 Language extensions

The αML language was designed as a minimal calculus for animating inductive definitions specified as schematic inference rules. Consequently, programs written in core αML can get rather large. The extra syntactic forms defined in Section 7.2 go some way to addressing this,

but programming in the language is still rather cumbersome. What is more, the unrestricted access to non-deterministic search features mean that execution can be inefficient. The addition of extra language features to αML could make the language more powerful and easier to use.

Polymorphic type system

The need for explicit type annotations can make code verbose, especially when writing down large inference rules. Also, the lack of built-in polymorphic datatypes such as lists means that these must be defined on a case-by-case basis. The addition of a polymorphic type system and type inference algorithm (Milner, 1978; Cardelli, 1987) would help to overcome these issues.

However, there are technical considerations: due to the nature of αML types we would need not only equality type variables à la Standard ML (Milner et al., 1997) but also type variables ranging over name sorts only. This could get rather complicated. Another potential problem is that the constraint transformation algorithm described in Chapter 6 relies on specific type information for all of the variables.

Pattern-matching syntax

Another major source of noise in αML programs is having to deconstruct complex data terms one constructor at a time. With suitable syntactic extensions, it might be possible to define an ML-style pattern-matching operation which could be compiled away into the existing simple deconstructors. We could go even further and include freshness information in patterns. For example, to pattern-match against a λ-binder (encoded using the nominal signature $\mathcal{F}$ from Figure 2.1) whilst simultaneously renaming the bound variable y to avoid all free variables of some terms t_1 and t_2, we might write the following.

$$\texttt{case}\ v\ \texttt{of Lam (}T\texttt{, <}y\texttt{>}t\texttt{) where}\ y\ \texttt{\#}\ \texttt{(}t_1\texttt{, }t_2\texttt{) ->}\cdots$$

The freshness annotations would be used by the system to produce new freshness constraints that would prevent name capture. The main difficulty would be coverage checking: in the current implementation of αML it is straightforward to check whether a `case` expression covers all of the constructors but that would be non-trivial with more complex patterns.

Programming in concrete syntax

The language currently forces programmers to write in abstract syntax, for example one must write `Lam<x>(Var x)` instead of the more natural $\lambda x.\,x$. Facilities for defining concrete syntax to go with datatype constructors would reduce the coding gap between informal mathematical definitions and executable αML code. Such facilities already exist in the `ott` system (Sewell et al., 2007). Writing an αML backend for `ott` would allow these features to be used in the definition of rules which could then be compiled into αML code.

Controlling non-determinism

The treatment of non-determinism and uninstantiated logic variables in αML is somewhat primitive. The distinction between rigid and flexible deconstructors is hard-wired into the semantics of the language and all αML code must account for the possibility that it could receive arguments which are only partially instantiated. This can lead to a proliferation of non-deterministic choice points, which in turn impacts on performance. A large amount of work has been done on controlling search and non-determinism in the context of the functional logic

language Curry (Braßel and Hanus, 2005; Hanus and Steiner, 2000) and some of these techniques could be applicable to αML.

Another interesting possibility would be to define additional control operators which control and encapsulate non-deterministic search computations (Hanus and Steiner, 1998; Braßel et al., 2004). As it stands, it is not possible to detect or handle failure in αML, so if a non-deterministic search fails to find a solution it may bring down the entire program. This is not always desirable: for example, given a binary transition relation REDUCE(t, t') we might like to define a predicate STUCK(t) which holds when there does not exist any term t' such that t reduces to t'. This predicate is defined in terms of negation and hence cannot be expressed directly in the language of α-inductive definitions presented in Chapter 2. This definition could be encoded in αML using a new construct not(e), which runs the computation e (of type prop) and which succeeds if e fails and fails if e succeeds. The STUCK predicate could then be defined by the following a single schematic rule.

```
 not(REDUCE(t,t'))
------------------
     STUCK(t)
```

Adding negation to αML could be technically difficult: the language of α-inductive definitions must be extended and it seems that the completeness result which we proved in Chapter 4 would no longer hold, as in Prolog. This is because αML programs may diverge, so not all unsatisfiable queries will fail finitely. The argument is similar to that from Section 5.6.

8.1.3 Improving the implementation

Static analyses such as mode and nondeterminism analyses could make αML programs more efficient, but there is still plenty of room for improvement in the implementation of the αML runtime. The runtime described in Appendix C was specifically designed for αML but the compiler does not perform any optimisations on the generated code. It might be worth exploring alternative implementation strategies such as compilation to an existing abstract machine like LLVM (Lattner and Adve, 2004) or a direct embedding in a source language such as OCaml or Haskell. Either of these would give us instant access to the high-performance optimising compiler of the target language.

However, the biggest opportunity for performance gains may come from parallelism. The evaluation of an αML branching expression $e \parallel e'$ seems ideally suited to parallel execution because the computations of e and e' are completely independent and never communicate with each other. One can envisage an αML implementation where the execution of different non-deterministic branches is shared between multiple processors, allowing them to execute in parallel. With the proliferation of multiple cores in modern machines, this could be a good way of increasing the performance of αML on larger problems.

The current implementation of αML in OCaml cannot easily take advantage of multicore machines because the OCaml runtime is entirely single-threaded. We could pursue this idea further by migrating the implementation to another language such as Haskell or F#.

8.1.4 Practical experience

Perhaps the most important way forward is to gain more experience of using αML in practice. There are many interesting and diverse systems ripe for implementation in αML, such as the pattern calculus (Jay and Kesner, 2006), ν-calculus (Pitts and Stark, 1993), various π-calculi (Milner, 1999; Sangiorgi and Walker, 2003) and multi-stage programming languages such as

$$\frac{}{n \triangleright (\alpha(n)\,,\top)} \qquad \frac{u \triangleright (t\,,\overline{c})}{\texttt{<}n\texttt{>}u \triangleright (\texttt{<}\alpha(n)\texttt{>}t\,,\overline{c})} \qquad \frac{u \triangleright (t\,,\overline{c})}{K\,u \triangleright (K\,t\,,\overline{c})} \qquad \frac{}{() \triangleright ((),\top)}$$

$$\frac{u_1 \triangleright (t_1\,,\overline{c}_1) \;\cdots\; u_n \triangleright (t_n\,,\overline{c}_n)}{(u_1\,,\ldots,u_n) \triangleright ((t_1\,,\ldots,t_n)\,,\overline{c}_1 \,\&\, \cdots \,\&\, \overline{c}_n)} \qquad \frac{}{\iota\,X \triangleright (\xi(X)\,,\top)}$$

$$\frac{\pi\,X \triangleright (y\,,\overline{c}) \qquad y \neq z \qquad y,z \in Var_{Temp}}{(n\,n')\pi\,X \triangleright (z\,,\overline{c} \,\&\, (\texttt{<}\alpha(n)\texttt{>}\texttt{<}\alpha(n')\texttt{>}y = \texttt{<}\alpha(n')\texttt{>}\texttt{<}\alpha(n)\texttt{>}z))}$$

Figure 8.1: Encoding of nominal unification terms

MetaML (Taha, 1999). We have also investigated the possibility of implementing a sequent-based theorem prover for first-order logic in αML.

The languages described above are mostly toy languages or small calculi: the application of αML to larger languages is currently hampered by practical limitations of the syntax and of the implementation. The extensions described in the previous sections could help to alleviate these problems and allow us to tackle larger systems such as Featherweight Java (Igarashi et al., 1999), and to extend our System F example to the full language with subtyping and records as featured in the POPLMARK challenge (Aydemir et al., 2005).

Integration with external tools such as a regression testing suite (for sanity checking revisions to a language definition), a model checker (for verifying simple meta-theoretic properties) or a graphical interface (to produce a visual aid for teaching proof theory, type systems and operational semantics) could also make the language more useful and powerful.

8.1.5 Constraint solving

There is much more work to be done on the theory of the constraint problem NonPermSat and the constraint solving procedure outlined in Chapter 6. In particular we hope to find a proof of Conjecture 6.3.2 which in turn would allow us to prove Conjecture 6.3.8, giving us a decision procedure for NonPermSat in terms of the the constraint transformation rules of Figure 6.1. We might approach this by proving the correctness of the termination checking procedure outlined in Section 7.3.

The algorithm used in the αML implementation only decides satisfiability of the constraint problem rather than enumerating the set of possible solutions, which might be more useful from a user's perspective. It should be straightforward to extend the code to do this, since the equivariant unification algorithm (Cheney, 2005a) can already enumerate solutions. Since the problem is known to be NP-complete it would also be worthwhile to investigate some heuristics to guide the search for satisfying valuations.

On a more theoretical note, it would be instructive to construct explicit reductions between NonPermSat and the various other constraint problems over nominal terms. We have already seen that a subset of NonPermSat can be encoded in nominal unification. Recalling the grammars of nominal unification terms and constraints from Section 6.4, we now present (without proof) an outline of how nominal unification could be encoded into NonPermSat.

We begin by partitioning the set of variables Var into three countably infinite sets: Var_{Name}, Var_{Uvar} and Var_{Temp}. We assume bijections $\alpha : Name \to Var_{Name}$ and $\xi : Uvar \to Var_{Uvar}$. The disjoint sets Var_{Name} and Var_{Uvar} are used to represent the names and unification variables of nominal unification, with a countably infinite supply of "spare" variables from Var_{Temp}.

The crux of the encoding is how we translate nominal terms into non-permutative terms. Suspended permutations are a major issue here because these do not exist in non-permutative terms. The rules in Figure 8.1 define a relation $u \triangleright (t\,, \overline{c})$ which translates a nominal unification term u into a non-permutative term t and an associated conjunction of non-permutative constraints which help to ensure that the translation is correct. The final two rules, which encode unknowns with suspended swappings, are the interesting ones. An unknown with the suspended identity permutation ($\iota\, X$) is translated directly as the appropriate variable $\xi(X) \in Var_{Uvar}$. If the suspension is not the identity, we use temporary variables from Var_{Temp} in the encoding. Care is needed to ensure that temporaries do not get re-used (for simplicity, these details are omitted from the rules in Figure 8.1). If we assume that $\pi\, X$ is translated to the temporary variable y then we translate $(n\, n')\pi\, X$ to the distinct temporary variable z, which is related to y via the new constraint

$$<\alpha(n)><\alpha(n')>y = <\alpha(n')><\alpha(n)>z.$$

This technique is familiar from the encoding of Mini-FreshML in Section 8.1.1 above.

With the encoding of terms u under our belt, the encoding of nominal unification constraints κ is fairly straightforward.

$$\begin{aligned} [\![u \approx ?\, u']\!] &\triangleq (t = t')\ \&\ \overline{c}\ \&\ \overline{c}' && \text{where } u \triangleright (t\,, \overline{c}) \text{ and } u' \triangleright (t'\,, \overline{c}'). \\ [\![n\ \#?\ u]\!] &\triangleq (\alpha(n)\ \#\ t)\ \&\ \overline{c} && \text{where } u \triangleright (t\,, \overline{c}). \end{aligned}$$

We encode a nominal unification problem $\overline{\kappa} = \{\kappa_1, \ldots, \kappa_n\}$ as the conjunction of the constraints produced by each constituent nominal unification constraint, with some extra freshness constraints to ensure that the variables representing names (i.e. those drawn from Var_{Name}) are mutually distinct. We write $names(\overline{\kappa})$ for the set of all names appearing in $\overline{\kappa}$, then:

$$[\![\overline{\kappa}]\!] \triangleq \#_{\alpha(names(\overline{\kappa}))}\ \&\ [\![\kappa_1]\!]\ \&\ \cdots\ \&\ [\![\kappa_n]\!].$$

Conjecture 8.1.1. *For any $\overline{\kappa}$, there exists a solution to $\overline{\kappa}$ under nominal unification precisely when the corresponding non-permutative constraint problem $[\![\overline{\kappa}]\!]$ is satisfiable.* ◇

We do not have a proof of Conjecture 8.1.1 but it seems reasonable, given the intuition behind the translation. We also expect that the set of satisfying valuations of the non-permutative constraint problem $[\![\overline{\kappa}]\!]$ can be related to the idempotent most general solution to $\overline{\kappa}$ computed by nominal unification.

We believe that a similar technique could be used to define a polynomial-time reduction of the more feature-rich equivariant unification problem into NonPermSat. Such a reduction must exist since NonPermSat and equivariant unification are both NP-complete. Furthermore, there must exist a polynomial-time reduction of NonPermSat into the problem of boolean satisfiability (SAT). Discovering this reduction would be of practical value: we could then implement a constraint solver by translating the constraint problem into a SAT problem and using one of the many mature, high-performance SAT-solvers. This might improve performance provided that the translation into SAT is not too expensive.

Finally, it would be interesting to investigate the relationship of our constraint problem to higher-order unification (Huet, 1975) and higher-order pattern unification (Dowek et al., 1996). The connection between nominal and higher-order pattern unification has already been explored in (Levy and Villaret, 2008; Cheney, 2005b).

8.2 Final remarks

We have reached the end of our tour of αML. Hopefully it has been both informative and entertaining. The work described in this dissertation was varied and enjoyable, but there is plenty more to be done. As declarative programming and formal methods gradually gain widespread use in computer science and industry, languages such as αML may become an important tool to language designers and to programmers in general. The work and ideas described in this dissertation may be a small step towards that ideal.

Bibliography

E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. An operational semantics for declarative multi-paradigm languages. In M. Comini and M. Falaschi, editors, *Proceedings of the* 11th *International Workshop on Functional and (Constraint) Logic Programming (WFLP 2002)*, volume 76 of *Electronic Notes in Theoretical Computer Science*, pages 1–19. Elsevier, 2002.

S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4): 776–822, 2000.

T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.

B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In G. C. Necula and P. Wadler, editors, *Proceedings of the* 35th *Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008)*, pages 3–15. ACM Press, 2008.

B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In J. Hurd and T. F. Melham, editors, *Proceedings of the* 18th *International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer-Verlag, 2005.

D. Baelde, A. Gacek, D. Miller, G. Nadathur, and A. Tiu. The Bedwyr system for model checking over syntactic expressions. In F. Pfenning, editor, *Proceedings of the* 21st *International Conference on Automated Deduction (CADE 2007)*, volume 4603 of *Lecture Notes in Computer Science*, pages 391–397. Springer-Verlag, 2007.

H. P. Barendregt. *The Lambda Calculus: its syntax and semantics*. North-Holland, revised edition, 1984.

A. Bauer. The Programming Language Zoo, 2008. Available at: `http://andrej.com/plzoo/`.

N. Benton. Machine obstructed proof. In 1st *Informal ACM SIGPLAN Workshop on Mechanizing Metatheory (WMM 2006)*, 2006.

S. Berghofer and C. Urban. Nominal inversion principles. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *Proceedings of the* 21st *International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2008)*, volume 5170 of *Lecture Notes in Computer Science*, pages 71–85. Springer-Verlag, 2008.

B. Boehm. A spiral model of software development and enhancement. *ACM SIGSOFT Software Engineering Notes*, 11(4):14–24, 1986.

B. Braßel and M. Hanus. Nondeterminism analysis of functional logic programs. In M. Gabbrielli and G. Gupta, editors, *Proceedings of the 21st International Conference on Logic Programming (ICLP 2005)*, volume 3668 of *Lecture Notes in Computer Science*, pages 265–279. Springer-Verlag, 2005.

B. Braßel, M. Hanus, and F. Huch. Encapsulating non-determinism in functional logic computations. *Journal of Functional and Logic Programming*, 2004(6), 2004.

W. E. Byrd and D. P. Friedman. alphaKanren: a fresh name in nominal logic programming. In D. Dubé, editor, *Proceedings of the 2007 Workshop on Scheme and Functional Programming*, pages 79–90, 2007. Université Laval technical report DIUL-RT-0701.

C. Calves and M. Fernández. Implementing nominal unification. In I. Mackie, editor, *Proceedings of the 3rd International Workshop on Term Graph Rewriting (TERMGRAPH 2006)*, volume 176 of *Electronic Notes in Theoretical Computer Science*, pages 25–37. Elsevier, 2007.

C. Calvès and M. Fernández. A polynomial nominal unification algorithm. *Theoretical Computer Science*, 403(2–3):285–306, 2008.

L. Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, 1987.

J. Cheney. The complexity of equivariant unification. In J. Díaz, J. Karhumäki, A. Lepistö, and D. Sannella, editors, *Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP 2004)*, volume 3142 of *Lecture Notes in Computer Science*, pages 332–344. Springer-Verlag, 2004a.

J. Cheney. Equivariant unification. In J. Giesl, editor, *Proceedings of the 16th International Conference on Rewriting Techniques and Applications (RTA 2005)*, volume 3467 of *Lecture Notes in Computer Science*, pages 74–89. Springer-Verlag, 2005a.

J. Cheney. *Nominal Logic Programming*. PhD thesis, Cornell University, 2004b.

J. Cheney. Relating nominal and higher-order pattern unification. In L. Vigneron, editor, *Proceedings of the 19th International Workshop on Unification (UNIF 2005)*, pages 104–119, 2005b. LORIA research report A05-R-022.

J. Cheney and C. Urban. Alpha-Prolog: a logic programming language with names, binding and alpha-equivalence. In B. Demoen and V. Lifschitz, editors, *Proceedings of the 20th International Conference on Logic Programming (ICLP 2004)*, number 3132 in Lecture Notes in Computer Science, pages 269–283. Springer-Verlag, 2004.

J. Cheney and C. Urban. Nominal logic programming. *ACM Transactions on Programming Languages and Systems*, 30(5):1–47, 2008.

S. Conchon, J.-C. Filliâtre, and J. Signoles. Designing a generic graph library using ML functors. In M. T. Morazán, editor, *Trends in Functional Programming, Volume 8*, pages 124–140. Intellect, 2008.

L. de Alfaro. Vec: extensible functional arrays for OCaml, 2008. Available at: `http://luca.dealfaro.org/`.

N. de Bruijn. Lambda calculus notation with nameless dummies: a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.

J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-order abstract syntax in Coq. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceeding of the* 2nd *International Conference on Typed Lambda Calculi and Applications (TLCA 1995)*, volume 902 of *Lecture Notes in Computer Science*, pages 124–138. Springer-Verlag, 1995.

G. Dowek. Higher-order unification and matching. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 16, pages 1009–1062. Elsevier, 2001.

G. Dowek, T. Hardin, C. Kirchner, and F. Pfenning. Higher-order unification via explicit substitutions: the case of higher-order patterns. In M. Maher, editor, *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming (JICSLP 1996)*, pages 259–723. MIT Press, 1996.

R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):83–110, 1992.

M. Felleisen and D. Friedman. Control operators, the SECD-machine, and the λ-calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. North-Holland, 1986.

C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1993)*, volume 28 of *ACM SIGPLAN Notices*, pages 237–247. ACM Press, 1993.

S. Frankau, D. Spinellis, N. Nassuphis, and C. Burgard. Going functional on exotic trades. *Journal of Functional Programming*, 19(1):27–45, 2009.

M. J. Gabbay. *A theory of inductive definitions with* α*-equivalence: semantics, implementation, programming language*. PhD thesis, University of Cambridge, 2000.

M. J. Gabbay and J. Cheney. A sequent calculus for nominal logic. In *Proceedings of the* 19th *Annual IEEE Symposium on Logic in Computer Science (LICS 2004)*, pages 139–148. IEEE Computer Society, 2004.

M. J. Gabbay and A. Mathijssen. Capture-avoiding substitution as a nominal algebra. *Formal Aspects of Computing*, 20(4–5):451–479, 2008.

M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13(3–5):341–363, 2002.

A. Gacek, D. Miller, and G. Nadathur. Reasoning in Abella about structural operational semantics specifications. In A. Abel and C. Urban, editors, *Proceedings of the* 3rd *International Workshop on Logical Frameworks and Metalanguages: Theory and Practice (LFMTP 2008)*, volume 228 of *Electronic Notes in Theoretical Computer Science*, pages 85–100. Elsevier, 2009.

J.-Y. Girard. *Proofs and Types*. Cambridge University Press, 1993.

R. L. Glass, I. Vessey, and V. Ramesh. Research in software engineering: an analysis of the literature. *Information and Software Technology*, 44(8):491–506, 2002.

W. D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13(2):225–230, 1981.

A. D. Gordon. Operational equivalences for untyped and polymorphic object calculi. Publications of the Newton Institute, pages 9–54. Cambridge University Press, 1998.

M. Hanus. Multi-paradigm declarative languages. In V. Dahl and I. Niemelä, editors, *Proceedings of the* 23rd *International Conference on Logic Programming (ICLP 2007)*, volume 4670 of *Lecture Notes in Computer Science*, pages 45–75. Springer-Verlag, 2007.

M. Hanus. A unified computation model for declarative programming. In M. Falaschi, M. Navarro, and A. Policriti, editors, *Proceedings of the 1997 Joint Conference on Declarative Programming (APPIA-GULP-PRODE 1997)*, pages 9–24, 1997.

M. Hanus and F. Steiner. Controlling search in declarative programs. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *Principles of Declarative Programming (Proceedings of the Joint International Symposium PLILP/ALP 1998)*, volume 1490 of *Lecture Notes in Computer Science*, pages 374–390. Springer-Verlag, 1998.

M. Hanus and F. Steiner. Type-based nondeterminism checking in functional logic programs. In *Proceedings of the* 2nd *International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2000)*, pages 202–213. ACM Press, 2000.

D. Herman and C. Flanagan. Status report: specifying JavaScript with ML. In C. Russo and D. Dreyer, editors, *Proceedings of the 2007 ACM Workshop on ML (ML 2007)*, pages 47–52. ACM Press, 2007.

C. A. R. Hoare. The Emperor's old clothes. *Communications of the ACM*, 24(2):75–83, 1981.

C. A. R. Hoare. Hints on programming language design. In *Conference Record of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1973)*. ACM Press, 1973.

D. J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.

G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1(1):27–57, 1975.

A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):132–146, 1999.

J. Jaffar, M. Maher, K. Marriott, and P. Stuckey. Semantics of constraint logic programming. *Journal of Logic Programming*, 37(1–3):1–46, 1998.

B. Jay and D. Kesner. Pure pattern calculus. In P. Sestoft, editor, *Proceedings of the* 15th *European Symposium on Programming (ESOP 2006)*, volume 3924 of *Lecture Notes in Computer Science*, pages 100–114. Springer-Verlag, 2006.

S. Kahrs. Mistakes and ambiguities in the Definition of Standard ML. Technical Report ECD-LFCS-93-257, University of Edinburgh, 1993.

E. E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Symposium on LISP and Functional Programming*, pages 151–161. ACM Press, 1986.

M. R. Lakin and A. M. Pitts. A metalanguage for structural operational semantics. In M. T. Morazán, editor, *Trends in Functional Programming, Volume 8*, pages 19–35. Intellect, 2008.

M. R. Lakin and A. M. Pitts. Resolving inductive definitions with binders in higher-order typed functional programming. In G. Castagna, editor, *Proceedings of the* 18th *European Symposium on Programming (ESOP 2009)*, volume 5502 of *Lecture Notes in Computer Science*, pages 47–61. Springer-Verlag, 2009.

P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

S. B. Lassen. Relational reasoning about contexts. Publications of the Newton Institute, pages 91–135. Cambridge University Press, 1998.

C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis and transformation. In *Proceedings of the* 2nd *International Symposium on Code Generation and Optimization (CGO 2004)*, pages 75–88. IEEE Computer Society, 2004.

X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

X. Leroy. A locally nameless solution to the POPLmark challenge. Technical Report 6098, INRIA, 2007.

J. Levy and M. Villaret. Nominal unification from a higher-order perspective. In A. Voronkov, editor, *Proceedings of the* 19th *International Conference on Rewriting Techniques and Applications (RTA 2008)*, volume 5117 of *Lecture Notes in Computer Science*, pages 246–260. Springer-Verlag, 2008.

H. G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *Conference Record of the* 17th *Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1990)*, pages 382–401. ACM Press, 1990.

I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):287–327, 1991.

J. Matthews, R. B. Findler, M. Flatt, and M. Felleisen. A visual environment for developing context-sensitive term rewriting systems. In V. van Oostrom, editor, *Proceedings of the 15th International Conference on Rewriting Techniques and Applications (RTA 2004)*, volume 3091 of *Lecture Notes in Computer Science*, pages 301–311. Springer-Verlag, 2004.

J. McKinna and R. Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3):373–409, 1999.

D. Miller and A. Tiu. A proof theory for generic judgments. *ACM Transactions on Computational Logic*, 6(4):749–783, 2005.

R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

R. Milner. A theory of type polymorphism in programming. *Journal of Computer Systems Science*, 17(3):348–375, 1978.

R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

G. Nadathur and D. Miller. An overview of λProlog. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the* 5th *International Conference on Logic Programming (ICLP 1988)*, pages 810–827. MIT Press, 1988.

J. P. Near, W. E. Byrd, and D. P. Friedman. alphaleanTAP: a declarative theorem prover for first-order classical logic. In M. G. de la Banda and E. Pontelli, editors, *Proceedings of the* 24th *International Conference on Logic Programming (ICLP 2008)*, volume 5366 of *Lecture Notes in Computer Science*, pages 238–252. Springer-Verlag, 2008.

R. Needham. Naming. In S. Mullender, editor, *Distributed Systems*, pages 89–101. ACM Press, 2nd edition, 1993.

P. Nickolas and P. J. Robinson. The Qu-Prolog unification algorithm: formalisation and correctness. *Theoretical Computer Science*, 169(1):81–112, 1996.

T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

S. Owens. A sound semantics for $\text{OCaml}_{\text{light}}$. In S. Drossopoulou, editor, *Proceedings of the* 17th *European Symposium on Programming (ESOP 2008)*, volume 4960 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2008.

C. H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.

M. S. Paterson and M. N. Wegman. Linear unification. In *Proceedings of the* 8th *Annual ACM Symposium on Theory of Computing (STOC 1976)*, pages 181–186. ACM Press, 1976.

E. Payet. Detecting non-termination of term rewriting systems using an unfolding operator. In G. Puebla, editor, *Proceedings of the* 16th *International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2006)*, volume 4407 of *Lecture Notes in Computer Science*, pages 194–209. Springer-Verlag, 2007.

S. L. Peyton-Jones and P. Wadler. Imperative functional programming. In *Conference Record of the* 20th *Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1993)*, pages 71–84. ACM Press, 1993.

S. L. Peyton-Jones, J.-M. Eber, and J. Seward. Composing contracts: an adventure in financial engineering (functional pearl). In *Proceedings of the* 5th *ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*, volume 35 of *ACM SIGPLAN Notices*, pages 280–292. ACM Press, 2000.

F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1988)*, volume 23 of *ACM SIGPLAN Notices*, pages 199–208. ACM Press, 1988.

F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the* 16th *International Conference on Automated Deduction (CADE 1999)*, volume 1632 of *Lecture Notes in Artifical Intelligence*, pages 202–206. Springer-Verlag, 1999.

A. M. Pitts. Alpha-structural recursion and induction. *Journal of the ACM*, 53(3):459–506, 2006.

A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186(2):165–193, 2003.

A. M. Pitts. Operational semantics and program equivalence. In *Applied Semantics, Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science*, pages 378–412. Springer-Verlag, 2002.

A. M. Pitts. Typed operational reasoning. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 7, pages 245–289. MIT Press, 2005.

A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Procceedings of the* 5th *International Conference on the Mathematics of Program Construction (MPC 2000)*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, 2000.

A. M. Pitts and M. R. Shinwell. Generative unbinding of names. *Logical Methods in Computer Science*, 4(1:4):1–33, 2008.

A. M. Pitts and I. D. B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In A. M. Borzyszkowski and S. Sokolowski, editors, *Proceedings of the* 18th *International Symposium on Mathematical Foundations of Computer Science (MFCS 1993)*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer-Verlag, 1993.

F. Pottier. An overview of Cαml. In N. Benton and X. Leroy, editors, *Proceedings of the 2005 ACM-SIGPLAN Workshop on ML (ML 2005)*, volume 148 of *Electronic Notes in Theoretical Computer Science*, pages 27–52. Elsevier, 2006.

F. Pottier. Static name control for FreshML. In *Proceedings of the* 22nd *Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 356–365. IEEE Computer Society, 2007.

D. Sangiorgi and D. Walker. *The Pi-Calculus: a theory of mobile processes*. Cambridge University Press, 2003.

S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In Z. Shao and B. C. Pierce, editors, *Proceedings of the* 36th *Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2009)*, pages 379–391. ACM Press, 2009.

P. Sewell, F. Zappa Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott: effective tool support for the working semanticist. In R. Hinze and N. Ramsey, editors, *Proceedings of the* 12th *ACM SIGPLAN International Conference on Functional Programming (ICFP 2007)*, pages 1–12. ACM Press, 2007.

M. R. Shinwell. Fresh O'Caml: Nominal abstract syntax for the masses. In N. Benton and X. Leroy, editors, *Proceedings of the 2005 ACM-SIGPLAN Workshop on ML (ML 2005)*, volume 148 of *Electronic Notes in Theoretical Computer Science*, pages 53–77. Elsevier, 2006.

M. R. Shinwell. *The Fresh Approach: functional programming with names and binders*. PhD thesis, University of Cambridge, 2005.

M. R. Shinwell and A. M. Pitts. On a monadic semantics for freshness. *Theoretical Computer Science*, 342(1):28–55, 2005.

M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: programming with binders made simple. In C. Runciman and O. Shivers, editors, *Proceedings of the* 8th *ACM SIGPLAN International Conference on Functional Programming (ICFP 2003)*, pages 263–274. ACM Press, 2003.

Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Functional Programming*, 29(1–3): 17–64, 1996.

W. Taha. *Multi-stage programming: its theory and applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.

A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

A. Tolmach, S. Antoy, and M. Nita. Implementing functional logic languages using multiple threads and stores. *ACM SIGPLAN Notices*, 39(9):90–102, 2004.

A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42:230–265, 1936.

C. Urban. Nominal Techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008.

C. Urban and J. Cheney. Avoiding equivariance in Alpha-Prolog. In P. Urzyczyn, editor, *Proceedings of the* 7th *International Conference on Typed Lambda Calculus and Applications (TLCA 2005)*, number 3461 in Lecture Notes in Computer Science, pages 74–89. Springer-Verlag, 2005.

C. Urban and T. Nipkow. Nominal verification of Algorithm W. Accepted for publication, 2008.

C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. *Theoretical Computer Science*, 323 (1–3):473–497, 2004.

C. Urban, S. Berghofer, and M. Norrish. Barendregt's variable convention in rule inductions. In F. Pfenning, editor, *Proceedings of the* 21st *International Conference on Automated Deduction (CADE 2007)*, volume 4603 of *Lecture Notes in Computer Science*, pages 35–50. Springer-Verlag, 2007.

C. Urban, J. Cheney, and S. Berghofer. Mechanizing the metatheory of LF. In *Proceedings of the* 23rd *IEEE Symposium on Logic in Computer Science (LICS 2008)*, pages 45–56. IEEE Computer Society, 2008.

J. B. Wells. Typability and type checking in the second order lambda-calculus are equivalent and undecidable. In *Proceedings of the* 9th *Annual IEEE Symposium on Logic in Computer Science (LICS 1994)*, pages 263–274. IEEE Computer Society, 1994.

N. Wirth. On the design of programming languages. In *Proceedings of 1974 IFIP Congress*, pages 386–393. North-Holland, 1974.

Appendices

Appendix A

Proof of compatibility

This appendix is devoted to building up a proof that $\cong^\circ$ is compatible, i.e. that

$$\Gamma \vdash e \mathrel{\widehat{\cong^\circ}} e' : T \implies \Gamma \vdash e \cong^\circ e' : T$$

for all Γ, e, e' and T. Since $\cong^\circ$ is known to be reflexive and substitutive, it suffices to prove this result for the special case where the expressions e and e' only contain variables of equality types, as we can use the substitutivity result to handle the closing substitution used to define $\cong^\circ$ in terms of $\cong$. Therefore we will assume that $\Delta \vdash e \mathrel{\widehat{\cong^\circ}} e' : T$ holds and attempt to prove that $\Delta \vdash e \cong^\circ e' : T$.

A.1 Proof outline

We use an operational proof technique similar to that of (Pitts and Shinwell, 2008), as opposed to the denotational approach of (Shinwell and Pitts, 2005). Like (Pitts and Shinwell, 2008) we use a variant of Howe's method (Howe, 1996). This proof strategy was originally developed to prove that applicative bisimilarity is a congruence in the lazy λ-calculus, but has proved to be a flexible, powerful tool in proofs about program equivalences in various settings, both lazy and eager.

Despite this, the technique itself and the reasons why it works are somewhat mysterious. For this reason, we prove our compatibility result by case analysis on the compatible extension $\widehat{\cong^\circ}$ of the operational equivalence relation, rather than by a single large termination induction like that used in (Pitts and Shinwell, 2008, Appendix A). Although more long-winded, our approach to proving compatibility is more didactic as it draws attention to the particular constructs of the αML syntax which require the full power of Howe's method for the proof to go through.

We proceed by cases on the possible forms of $\Delta \vdash e \mathrel{\widehat{\cong^\circ}} e' : T$, by inspection of the compatible refinement rules from Figure 5.1. As we shall see, the most challenging case is that for recursive function values. That case, and the one for `let` bindings, require the most work and are deferred to Section A.3 and Section A.2, respectively.

The remaining cases go through using the rules from Figure 5.1, and we dispense with these now. The cases for variables x (of an equality type) and `T` values are straightforward, using the appropriate compatible refinement rule and the fact that $\cong^\circ$ is reflexive. We present the case for `case` expressions in full as it is the only non-trivial one. We also give details of the case for abstractions because the remaining cases follow this template.

- $\boldsymbol{\Delta \vdash (\texttt{case}\, v \,\texttt{of}\, K_1\, x_1 \texttt{->} e_1 \mid \cdots \mid K_n\, x_n \texttt{->} e_n) \mathrel{\widehat{\cong^\circ}}}$
  $\boldsymbol{(\texttt{case}\, v' \,\texttt{of}\, K_1\, x_1 \texttt{->} e'_1 \mid \cdots \mid K_n\, x_n \texttt{->} e'_n) : T.}$]

By using the compatible refinement rule for case expressions on the above assumption we get that $D = K_1 \texttt{ of } T_1 \mid \cdots \mid K_n \texttt{ of } T_n$, $x_1, \ldots, x_n \notin dom(\Delta)$, $\Delta \vdash v \cong^\circ v' : D$ and $\forall i \in \{1, \ldots, n\}$. $\Delta, x_i : T_i \vdash e_i \cong^\circ e'_i : T$ all hold, where the variables $x_1, \ldots, x_n$ are also mutually distinct. Now, the values v and v' in the $\cong^\circ$-judgement could each either be variables or have a data terms $K\, v^*$, for some v^*. Up to symmetry, there are three cases to consider.

Case $v = x$ and $v' = x'$. By the form of Δ it follows that $\Delta(x) = \Delta(x') = S$, i.e. that the data sort D is actually a nominal data sort S, where $\texttt{datatype } S =_\Sigma K_1 \texttt{ of } E_1 \mid \cdots \mid K_n \texttt{ of } E_n$ for some $K_1, \ldots, K_n$ and $E_1, \ldots, E_n$. Then, we have that $\Delta, x_i : E_i \vdash e_i \cong^\circ e'_i : T$ holds for all $i \in \{1, \ldots, n\}$, and also that $\Delta \vdash x \cong^\circ x' : D$ holds. Let Δ', $\bar{c}$, F and T' be such that $\Delta' \supseteq \Delta$, $\Delta' \vdash F : T \to T'$ and $\Delta' \vdash \bar{c} : \texttt{prop}$ all hold. Then, we can show that $\Delta', x_j : E_j \vdash \bar{c} \,\&\, x = K_j\, x_j : \texttt{prop}$ holds for any $j \in \{1, \ldots, n\}$, and by operational equivalence we know that

$$\exists \Delta', x_j : E_j (\bar{c} \,\&\, x = K_j\, x_j; F; e_j)\downarrow \iff \exists \Delta', x_j : E_j (\bar{c} \,\&\, x = K_j\, x_j; F; e'_j)\downarrow.$$

We can rewrite these configurations (using rule (I4)) to give

$$\exists \Delta' (\bar{c}; F; \texttt{case } x \texttt{ of } K_1\, x_1 \texttt{->} e_1 \mid \cdots \mid K_n\, x_n \texttt{->} e_n)\downarrow \iff \exists \Delta' (\bar{c}; F; \texttt{case } x \texttt{ of } K_1\, x_1 \texttt{->} e'_1 \mid \cdots \mid K_n\, x_n \texttt{->} e'_n)\downarrow,$$

since both configurations terminate if $\models \exists \Delta', x_j : E_j (\bar{c} \,\&\, x = K_j\, E_j)$ hold and neither terminates if the constraint is unsatisfiable. Then since x and x' are operationally equivalent, we can use an intermediate frame stack to show that

$$\exists \Delta' (\bar{c}; F; \texttt{case } x \texttt{ of } K_1\, x_1 \texttt{->} e_1 \mid \cdots \mid K_n\, x_n \texttt{->} e_n)\downarrow \iff \exists \Delta' (\bar{c}; F; \texttt{case } x' \texttt{ of } K_1\, x_1 \texttt{->} e'_1 \mid \cdots \mid K_n\, x_n \texttt{->} e'_n)\downarrow.$$

Since $v = x$ and $v' = x'$, the result follows.

Case $v = x$ and $v' = K_j\, v^\dagger$. As before, it must be the case that D is actually a nominal data sort S, where $\texttt{datatype } S =_\Sigma K_1 \texttt{ of } E_1 \mid \cdots \mid K_n \texttt{ of } E_n$ for some $K_1, \ldots, K_n$ and some $E_1, \ldots, E_n$. Then, we have that $\Delta, x_i : E_i \vdash e_i \cong^\circ e'_i : T$ holds for all $i \in \{1, \ldots, n\}$, and also that $\Delta \vdash v^\dagger : E_j$ and $\Delta \vdash x \cong^\circ K_j\, v^\dagger : S$. Let Δ', $\bar{c}$, F and T' be such that $\Delta' \supseteq \Delta$, $\Delta' \vdash \bar{c} : \texttt{prop}$ and $\Delta' \vdash F : T \to T'$ all hold. Since $\cong^\circ$ is reflexive and substitutive, we know that $\Delta \vdash e_j[v^\dagger / x_j] \cong^\circ e'_j[v^\dagger / x_j] : T$, i.e. that

$$\exists \Delta' (\bar{c}; F; e_j[v^\dagger / x_j])\downarrow \iff \exists \Delta' (\bar{c}; F; e'_j[v^\dagger / x_j])\downarrow.$$

By rules (I1) and (P4) we get that

$$\exists \Delta' (\bar{c}; F; \texttt{case } K_j\, v^\dagger \texttt{ of } K_1\, x_1 \texttt{->} e_1 \mid \cdots \mid K_n\, x_n \texttt{->} e_n)\downarrow \iff \exists \Delta' (\bar{c}; F; \texttt{case } K_j\, v^\dagger \texttt{ of } K_1\, x_1 \texttt{->} e'_1 \mid \cdots \mid K_n\, x_n \texttt{->} e'_n)\downarrow$$

holds. Then, since x and $K_j\, v^\dagger$ are operationally equivalent, we get

$$\exists \Delta' (\bar{c}; F; \texttt{case } x \texttt{ of } K_1\, x_1 \texttt{->} e_1 \mid \cdots \mid K_n\, x_n \texttt{->} e_n)\downarrow \iff \exists \Delta' (\bar{c}; F; \texttt{case } K_j\, v^\dagger \texttt{ of } K_1\, x_1 \texttt{->} e'_1 \mid \cdots \mid K_n\, x_n \texttt{->} e'_n)\downarrow.$$

The result follows because $v = x$ and $v' = K_j\, v^\dagger$.

Case $v = K_j\, v^*$ and $v' = K_j\, v^\dagger$. We observe that the two values v and v' must have the same data constructor at their root (here K_j). This must be the case because if the constructors were different then v and v' would not be operationally equivalent: they could be distinguished using a `case` expression which terminates in the branch corresponding to one constructor but diverges in the branch corresponding to the other. Therefore, we have $\Delta \vdash K_j\, v^* \cong^\circ K_j\, v^\dagger : D$, $\Delta \vdash v^* : T_j$ and $\Delta \vdash v^\dagger : T_j$.

It must also be the case that $\Delta \vdash v^* \cong^\circ v^\dagger : T_j$ holds, because otherwise we could distinguish $K_j\, v^*$ and $K_j\, v^\dagger$ by removing their outermost constructor and placing the values into a configuration that distinguishes v^* and $v^\dagger$. Now, since $\cong^\circ$ is substitutive we get that $\Delta \vdash e_j[v^*/x_j] \cong^\circ e'_j[v^\dagger/x_j] : T$ holds, and we then pick arbitary Δ', $\overline{c}$, F and T' such that $\Delta' \supseteq \Delta$, $\Delta \vdash \overline{c} : \texttt{prop}$ and $\Delta' \vdash F : T \to T'$ all hold. Then, by the definition of operational equivalence we have

$$\exists \Delta' (\overline{c}; F; e_j[v^*/x_j]) \downarrow \iff \exists \Delta' (\overline{c}; F; e'_j[v^\dagger/x_j]) \downarrow.$$

By rules (I1) and (P4), this can be rewritten to give

$$\begin{gathered}\exists \Delta' (\overline{c}; F; \texttt{case}\, K_j\, v^*\, \texttt{of}\, K_1\, x_1 \texttt{->} e_1 \mid \cdots \mid K_n\, x_n \texttt{->} e_n) \downarrow \iff \\ \exists \Delta' (\overline{c}; F; \texttt{case}\, K_j\, v^\dagger\, \texttt{of}\, K_1\, x_1 \texttt{->} e'_1 \mid \cdots \mid K_n\, x_n \texttt{->} e'_n) \downarrow.\end{gathered}$$

Finally, since $v = K_j\, v^*$ and $v' = K_j\, v^\dagger$, we have the desired result.

- **$\Delta \vdash \texttt{<}v_1\texttt{>}v_2 \mathrel{\widehat{\cong^\circ}} \texttt{<}v'_1\texttt{>}v'_2 : \texttt{[}N\texttt{]}\, E$.**

  By assumption we have $\Delta \vdash \texttt{<}v_1\texttt{>}v_2 \mathrel{\widehat{\cong^\circ}} \texttt{<}v'_1\texttt{>}v'_2 : \texttt{[}N\texttt{]}\, E$, and using the rule for abstractions from Figure 5.1 we get that $\Delta \vdash v_1 \cong^\circ v'_1 : N$ and $\Delta \vdash v_2 \cong^\circ v'_2 : E$ both hold, for some N, E. The only values of name sort are variables which implies that $v_1 = v'_1 = x$, for some $x \in dom(\Delta)$ where $\Delta(x) = N$ (if v_1 and v'_1 were distinct variables, we could construct a configuration that distinguished them, thereby contradicting $\Delta \vdash v_1 \cong^\circ v'_1 : N$). Then, for any Δ', F, $\overline{c}$ and T' such that $\Delta' \supseteq \Delta$, $\Delta' \vdash F : \texttt{[}N\texttt{]}\, E \to T'$ and $\Delta' \vdash \overline{c} : \texttt{prop}$, by operational equivalence we have that

$$\exists \Delta' (\overline{c}; F \circ (z.\, \texttt{<}x\texttt{>}z); v_2) \downarrow \iff \exists \Delta' (\overline{c}; F \circ (z.\, \texttt{<}x\texttt{>}z); v'_2) \downarrow,$$

  where $z \notin dom(\Delta')$ is a freshly-chosen variable. Using rules (I1) and (P1), and the fact that $v_1 = v'_1 = x$, to get

$$\exists \Delta' (\overline{c}; F; \texttt{<}v_1\texttt{>}v_2) \downarrow \iff \exists \Delta' (\overline{c}; F; \texttt{<}v'_1\texttt{>}v'_2) \downarrow,$$

  which is equivalent to $\Delta \vdash \texttt{<}v_1\texttt{>}v_2 \cong^\circ \texttt{<}v'_1\texttt{>}v'_2 : \texttt{[}N\texttt{]}\, E$, as required.

The other cases go through by induction: each time, we must choose appropriate evaluation contexts for the various syntactic constructs of αML.

A.2 Proof of compatibility: `let` bindings

In this section we will prove the case of the compatibility theorem for `let` bindings. The main lemma in this case is that two $\widehat{\cong^\circ}$-related frame stacks may be interchanged in a configuration without affecting its termination behaviour.

Lemma A.2.1. *For all $n \geq 0$, if $\Delta \vdash \overline{c} : \texttt{prop}$, $\Delta \vdash e : T$, $\Delta \vdash F \mathrel{\widehat{\cong^\circ}} F' : T \to T'$ and $\exists \Delta (\overline{c}; F; e) \downarrow^n$ all hold then $\exists \Delta (\overline{c}; F'; e) \downarrow$ holds also.*

Proof. The proof is by induction on n, and the cases are as follows.

Base case. In this case, we know that $\exists\Delta(\overline{c}; F; e)\downarrow^0$, from which we get that $e \in Val_\Sigma$, $F = \mathtt{Id}$ and $\models \exists\Delta(\overline{c})$ all hold. Then, by the compatible refinement rule for empty frame stacks we know that $F' = \mathtt{Id}$ also, and thus we get that $\exists\Delta(\overline{c}; F'; e)\downarrow$ holds, as required.

Induction step. We know that $\exists\Delta(\overline{c}; F; e)\downarrow^{n+1}$ holds, which means that the reduction sequence is at least one step long. We proceed by a case split on the first $\longrightarrow$ rule used in this derivation. We only present the cases for rules which actually manipulate the frame stack: the other cases go through straightforwardly by induction on n.

(I1) and (P1). In this case we know that $F = F_1 \circ (x.\, e_1)$ and $e = v$, for some F_1, x, e_1, v. Furthermore, by the rule for non-empty frame stacks from Figure 5.2 we know that $F' = F_2 \circ (x.\, e_2)$, for some F_2 and e_2, where $\Delta \vdash F_1 \mathrel{\widehat{\cong^\circ}} F_2 : T'' \to T'$ and $\Delta, x : T \vdash e_1 \cong^\circ e_2 : T''$ both hold, for some T''. By applying rules (I1) and (P1) we get that $\exists\Delta(\overline{c}; F_1; e_1[v/x])\downarrow^n$, and using our induction hypothesis we get that $\exists\Delta(\overline{c}; F_2; e_1[v/x])\downarrow$. Since e_1 and e_2 are operationally equivalent and $\cong^\circ$ is substitutive we get that $\exists\Delta(\overline{c}; F_2; e_2[v/x])\downarrow$ holds, and using (I1) and (P1) we get $\exists\Delta(\overline{c}; F_2 \circ (x.\, e_2); v)\downarrow$, i.e. $\exists\Delta(\overline{c}; F'; e)\downarrow$, as required.

(I1) and (P2). In this case we know that $e = \mathtt{let}\, x = e_1\ \mathtt{in}\ e_2$, for some x, e_1, e_2. Since $\Delta \vdash e : T$ we get that $\Delta \vdash e_1 : T''$ and $\Delta, x : T'' \vdash e_2 : T$ both hold, for some T''. Then, since $\cong^\circ$ is reflexive we have $\Delta, x : T'' \vdash e_2 \cong^\circ e_2 : T$. We can then show that $\Delta \vdash (F \circ (x.\, e_2)) \mathrel{\widehat{\cong^\circ}} (F' \circ (x.\, e_2)) : T \to T'$ holds. By applying rules (I1) and (P2) to the original termination judgement we get that $\exists\Delta(\overline{c}; F \circ (x.\, e_2); e_1)\downarrow^n$ holds, and by induction we can show that $\exists\Delta(\overline{c}; F' \circ (x.\, e_2); e_1)\downarrow$. It then follows that $\exists\Delta(\overline{c}; F'; \mathtt{let}\, x = e_1\ \mathtt{in}\ e_2)\downarrow$, i.e. that $\exists\Delta(\overline{c}; F'; e)\downarrow$, as required.

This completes the proof of Lemma A.2.1. ☐

We can now prove the main result of this section, which gives us the compatibility result in the case of a `let` binding.

Lemma A.2.2. *For any two* `let` *bindings* $\mathtt{let}\, x = e_1\ \mathtt{in}\ e_2$ *and* $\mathtt{let}\, x = e_1'\ \mathtt{in}\ e_2'$*:*

if $\quad \Delta \vdash (\mathtt{let}\, x = e_1\ \mathtt{in}\ e_2) \mathrel{\widehat{\cong^\circ}} (\mathtt{let}\, x = e_1'\ \mathtt{in}\ e_2') : T$

then $\quad \Delta \vdash (\mathtt{let}\, x = e_1\ \mathtt{in}\ e_2) \cong^\circ (\mathtt{let}\, x = e_1'\ \mathtt{in}\ e_2') : T.$

Proof. We assume that $\Delta \vdash (\mathtt{let}\, x = e_1\ \mathtt{in}\ e_2) \mathrel{\widehat{\cong^\circ}} (\mathtt{let}\, x = e_1'\ \mathtt{in}\ e_2') : T$, from which we get that $\Delta \vdash e_1 \cong^\circ e_1' : T'$ and $\Delta, x : T' \vdash e_2 \cong^\circ e_2' : T$ both hold, for some T' and where $x \notin dom(\Delta)$. We now pick any Δ', $\overline{c}$, F and T'', and assume that $\Delta' \supseteq \Delta$, $\Delta' \vdash F : T \to T''$ and $\Delta' \vdash \overline{c} : \mathtt{prop}$ all hold. Since $\cong^\circ$ is reflexive, we can use Lemma 5.2.2 to deduce that $\widehat{\cong^\circ}$ is also reflexive, and then we get that $\Delta' \vdash F \mathrel{\widehat{\cong^\circ}} F : T \to T''$. By the compatible refinement rule for non-empty frame stacks we get that $\Delta' \vdash F \circ (x.\, e_2) \mathrel{\widehat{\cong^\circ}} F \circ (x.\, e_2') : T' \to T''$. Using Lemma A.2.1 we get that

$$\exists\Delta'(\overline{c}; F \circ (x.\, e_2); e_1')\downarrow \iff \exists\Delta'(\overline{c}; F \circ (x.\, e_2'); e_1')\downarrow$$

since $\widehat{\cong^\circ}$ is symmetric (by a similar argument to reflexivity, using Lemma 5.2.2). By operational equivalence we have $\exists\Delta'(\overline{c}; F \circ (x.\, e_2); e_1)\downarrow \iff \exists\Delta'(\overline{c}; F \circ (x.\, e_2); e_1')\downarrow$ and combining the two using transitivity yields $\exists\Delta'(\overline{c}; F \circ (x.\, e_2); e_1)\downarrow \iff \exists\Delta'(\overline{c}; F \circ (x.\, e_2'); e_1')\downarrow$. We can rewrite this using rules (I1) and (P2) to give

$$\exists\Delta'(\overline{c}; F; \mathtt{let}\, x = e_1\ \mathtt{in}\ e_2)\downarrow \iff \exists\Delta'(\overline{c}; F; \mathtt{let}\, x = e_1'\ \mathtt{in}\ e_2')\downarrow$$

so we have $\Delta \vdash (\mathtt{let}\, x = e_1\ \mathtt{in}\ e_2) \cong^\circ (\mathtt{let}\, x = e_1'\ \mathtt{in}\ e_2') : T$, as required. ☐

A.3 Proof of compatibility: recursive functions

This case of the compatibility proof is the most difficult. For an intuition as to why this is so, suppose that we take two operationally-equivalent expressions e and e' and package them up in two recursive functions $\texttt{fun}\, f\,\texttt{(}x : T_{in}\texttt{)} : T_{out} = e$ and $\texttt{fun}\, f\,\texttt{(}x : T_{in}\texttt{)} : T_{out} = e'$. These functions are values which could be passed around the program in any way at all. Eventually they may be applied to an argument, but we have no idea when (or if) this will happen. Therefore the evaluation of e (or e') could happen in an arbitrary context. Therefore this case calls on the power of Howe's method (Howe, 1996). The fact that functions in αML are always recursive causes no additional difficulty on top of that caused by the presence of normal λ-abstraction.

We begin by defining the following auxiliary relation, which we will use in the proof.

$$(\mathcal{E}^*\text{-}\textsc{Fun})\ \frac{\Delta, f : T_{in} \to T_{out}, x : T_{in} \vdash e \cong^\circ e' : T_{out}}{\Delta \vdash (\texttt{fun}\, f\,\texttt{(}x : T_{in}\texttt{)} : T_{out} = e)\ \mathcal{E}^*\ (\texttt{fun}\, f\,\texttt{(}x : T_{in}\texttt{)} : T_{out} = e') : T_{in} \to T_{out}}$$

$$(\mathcal{E}^*\text{-}\textsc{Howe})\ \frac{\Gamma \vdash e\ \widehat{\mathcal{E}^*}\ e'' : T \qquad \Gamma \vdash e'' \cong^\circ e' : T}{\Gamma \vdash e\ \mathcal{E}^*\ e' : T}$$

In making this definition we rely implicitly on the fact that the compatible refinement operator is monotone (i.e. that if $\mathcal{E}_1 \subseteq \mathcal{E}_2$ then $\widehat{\mathcal{E}_1} \subseteq \widehat{\mathcal{E}_2}$). This is a special case of Howe's construction, which closes under compatible refinement whilst also composing with operational equivalence on the right in rule ($\mathcal{E}^*$-Howe). This trick is crucial: it ensures that two $\mathcal{E}^*$-related expressions do not need to be syntactically similar.

The proof strategy in this section can be summarised as follows: we show that $\mathcal{E}^*$ is compatible then prove that it coincides with operational equivalence. This allows us to deduce the desired result about recursive functions because rule ($\mathcal{E}^*$-Fun) ensures that the $\mathcal{E}^*$ relation contains the recursive function values that we are interested in, namely those whose bodies are $\cong^\circ$-related.

Lemma A.3.1 ($\mathcal{E}^*$ is compatible). *If $\Gamma \vdash e\ \widehat{\mathcal{E}^*}\ e' : T$ then $\Gamma \vdash e\ \mathcal{E}^*\ e' : T$.*

Proof. This follows by ($\mathcal{E}^*$-Howe) because $\cong^\circ$ is reflexive. □

We begin by showing that $\mathcal{E}^*$ is reflexive and has the weakening property, and use this to show that operational equivalence is contained within $\mathcal{E}^*$. This direction is actually rather straightforward: the difficulty comes in showing the reverse containment.

Lemma A.3.2. *$\mathcal{E}^*$ is reflexive, and has the weakening property, i.e. if $\Gamma \vdash e\ \mathcal{E}^*\ e' : T$ and $\Gamma' \supseteq \Gamma$ then $\Gamma' \vdash e\ \mathcal{E}^*\ e' : T$.*

Proof. We prove both by induction over the the structure of e. To show reflexivity we use Lemma A.3.1, and for weakening we use the fact that $\cong^\circ$ has the weakening property (Theorem 5.3.1). □

Lemma A.3.3 ($\cong^\circ \subseteq \mathcal{E}^*$). *If $\Gamma \vdash e \cong^\circ e' : T$ then $\Gamma \vdash e\ \mathcal{E}^*\ e' : T$.*

Proof. This follows from the fact that $\widehat{\mathcal{E}^*}$ is reflexive (by Lemma 5.2.2 and Lemma A.3.2). We can then use ($\mathcal{E}^*$-Howe) to derive the result. □

We now begin to prove some preparatory results about the $\mathcal{E}^*$ relation, as we build up to the key lemma in this proof.

Lemma A.3.4. *If* $\Gamma \vdash e_1\, \mathcal{E}^*\, e_2 : T$ *and* $\Gamma \vdash e_2 \cong^\circ e_3 : T$ *then* $\Gamma \vdash e_1\, \mathcal{E}^*\, e_3 : T$.

Proof. A case split is required on how the $\mathcal{E}^*$-judgement was derived—if ($\mathcal{E}^*$-FUN) was used we need Lemma A.3.3, but if ($\mathcal{E}^*$-HOWE) was used we require the fact that $\cong^\circ$ is transitive. □

Lemma A.3.5. *If* $\Gamma \vdash v\, \mathcal{E}^*\, e' : T$ *then there exists a value* v' *such that* $\Gamma \vdash v\, \mathcal{E}^*\, v' : T$ *and* $\Gamma \vdash v' \cong^\circ e' : T$.

Proof. By a straightforward case analysis on v. In the case where v is a recursive function, a case split is required on whether the $\mathcal{E}^*$-judgement was derived using ($\mathcal{E}^*$-FUN) or ($\mathcal{E}^*$-HOWE). □

We now show that $\mathcal{E}^*$ is preserved by the capture-avoiding substitution operation.

Lemma A.3.6 ($\mathcal{E}^*$ is substitutive). *If* $\Gamma, x : T \vdash e_1\, \mathcal{E}^*\, e_2 : T'$, $\Gamma \vdash v_1\, \mathcal{E}^*\, v_2 : T$ *and* $x \notin dom(\Gamma)$, *then* $\Gamma \vdash e_1[v_1/x]\, \mathcal{E}^*\, e_1[v_2/x] : T'$.

Proof. By a lengthy induction on the structure of e_1. □

Showing that $\mathcal{E}^*$ is contained within $\cong^\circ$ for certain restricted classes of αML expressions at certain types is actually not that difficult. We dispense with two important sub-cases now, for values of equality types and for constraints.

Lemma A.3.7 ($\mathcal{E}^* \subseteq \cong^\circ$ for values of equality type). *If* $\Gamma \vdash v\, \mathcal{E}^*\, v' : E$ *then* $\Gamma \vdash v \cong^\circ v' : E$.

Proof. By induction on the structure of v, using the compatibility results which we have already established for values of equality type (i.e. if $\Gamma \vdash v\, \widehat{\cong^\circ}\, v' : E$ then $\Gamma \vdash v \cong^\circ v' : E$). □

Lemma A.3.8 ($\mathcal{E}^* \subseteq \cong^\circ$ for constraints). *If* $\Gamma \vdash c\, \mathcal{E}^*\, c' : \texttt{prop}$ *then* $\Gamma \vdash c \cong^\circ c' : \texttt{prop}$.

Proof. We assume $\Gamma \vdash c\, \mathcal{E}^*\, c' : \texttt{prop}$, which must have been derived using ($\mathcal{E}^*$-HOWE)—hence we know that $\Gamma \vdash c\, \widehat{\mathcal{E}^*}\, e : \texttt{prop}$ and $\Gamma \vdash e \cong^\circ c' : \texttt{prop}$ for some e. Furthermore, we know that the former must have been derived using the compatible refinement rule for an equality or freshness constraint, so e must also be a constraint. We case split on whether the rule for equality or freshness constraints was used to show this, and invoke Lemma A.3.7 once we have picked apart the constraint using the appropriate rule. □

The Howe's method technique used in (Pitts and Shinwell, 2008) did not need to take constraint satisfaction into account because the FreshML language does not include constraints. However, we must deal with constraints and the following lemmas relate the satisfiability of constraints to their termination behaviour when they are evaluated in αML. We first fix a representation of a constraint $\overline{c}$ in terms of a frame stack $F_{\overline{c}}$. The rationale is to sequentially process all of the constraints in $\overline{c}$. A frame stack is ideal for this because it specifies the appropriate continuation after each individual constraint has been processed. This device is only used in this section of the proof.

Definition A.3.9 (Stack representation of constraint problems). Given a constraint $\overline{c} = c_1\, \texttt{\&} \cdots \texttt{\&}\, c_n$, we define a frame stack representation

$$F_{\overline{c}} \quad \triangleq \quad \texttt{Id} \circ (x_n.\, c_n) \circ \cdots \circ (x_1.\, c_1)$$

where the $x_1, \ldots, x_n$ are distinct variables which do not appear in the corresponding constraints $c_1, \ldots, c_n$. ◇

We now show relate the satisfiability of a constraint $\overline{c}$ to the termination behaviour of its frame stack representation $F_{\overline{c}}$ (when run in the empty context).

Lemma A.3.10. $\exists\Delta(\emptyset; F_{\overline{c}}; \mathsf{T})\downarrow$ *iff* $\models \exists\Delta(\overline{c})$.

Proof. This result follows from Theorem 4.3.10. □

We now prove the main lemma regarding constraint satisfaction required for this section, which is that if two constraints are related by $\widehat{\mathcal{E}^*}$ (the compatible refinement of $\mathcal{E}^*$) then they are either both satisfiable or both unsatisfiable.

Lemma A.3.11. *If* $\Delta \vdash \overline{c}\,\widehat{\mathcal{E}^*}\,\overline{c}' : \mathtt{prop}$ *then* $\models \exists\Delta(\overline{c})$ *iff* $\models \exists\Delta(\overline{c}')$.

Proof. By assumption we have $\Delta \vdash \overline{c}\,\widehat{\mathcal{E}^*}\,\overline{c}' : \mathtt{prop}$. From the rule for constraint problems from Figure 5.2 we get that $\overline{c} = c_1 \;\&\; \cdots \;\&\; c_n$ and $\overline{c}' = c'_1 \;\&\; \cdots \;\&\; c'_n$, for some n, and that $\Delta \vdash c_i\,\mathcal{E}^*\,c'_i : \mathtt{prop}$ for all $i \in \{1, \ldots, n\}$. Then we can show that $\Delta \vdash F_{\overline{c}}\,\widehat{\mathcal{E}^*}\,F_{\overline{c}'} : \mathtt{prop} \to \mathtt{prop}$ holds, and by Lemma A.3.10 we know that $\exists\Delta(\emptyset; F_{\overline{c}}; \mathsf{T})\downarrow \iff \models \exists\Delta(\overline{c})$ and $\exists\Delta(\emptyset; F_{\overline{c}'}; \mathsf{T})\downarrow \iff \models \exists\Delta(\overline{c}')$ both hold. Furthermore, since $F_{\overline{c}}$ and $F_{\overline{c}'}$ are $\widehat{\cong^\circ}$-related, we can use Lemma A.2.1 to show that $\exists\Delta(\emptyset; F_{\overline{c}}; \mathsf{T})\downarrow \iff \exists\Delta(\emptyset; F_{\overline{c}'}; \mathsf{T})\downarrow$. Thus we get that $\models \exists\Delta(\overline{c}) \iff \models \exists\Delta(\overline{c}')$, as required. □

We can now state and prove the central lemma in this section, concerning the termination behaviour of $\mathcal{E}^*$-related expressions.

Lemma A.3.12 (Termination of $\mathcal{E}^*$-related expressions). *For all $n \geq 0$, it is the case that*

$$(\Delta \vdash e\,\mathcal{E}^*\,e' : T \wedge \Delta \vdash \overline{c}\,\widehat{\mathcal{E}^*}\,\overline{c}' : \mathtt{prop} \wedge \Delta \vdash F\,\widehat{\mathcal{E}^*}\,F' : T \to T' \wedge \exists\Delta(\overline{c}; F; e)\downarrow^n) \implies \exists\Delta(\overline{c}'; F'; e')\downarrow.$$

Proof. The proof is by induction on n—the cases are as follows.

Base case. In this case, we know that $\exists\Delta(\overline{c}; F; e)\downarrow^0$, from which we know that $e = v$ (for some v), $\models \exists\Delta(\overline{c})$ and $F = \mathtt{Id}$. By the compatible refinement rule for empty frame stacks we also know that $F' = \mathtt{Id}$. From Lemma A.3.5 we get that there exists v' such that $\Delta \vdash v\,\mathcal{E}^*\,v' : T$ and $\Delta \vdash v' \cong^\circ e' : T$ both hold. Using Lemma A.3.11 we get that $\models \exists\Delta(\overline{c}')$, and then by the definition of termination we have $\exists\Delta(\overline{c}'; \mathtt{Id}; v')\downarrow$. Finally, by operational equivalence we can show that $\exists\Delta(\overline{c}'; F'; e')\downarrow$, as required.

Induction step. We know that $\exists\Delta(\overline{c}; F; e)\downarrow^{n+1}$, and proceed by a case split on the first rule used in this derivation. The cases for (I1) in conjunction with the pure rules (P1), (P2), (P4) and (P5), as well as for rules (I3)–(I6) are fairly standard inductions and so are omitted: the case for (I1) and (P1) requires Lemma A.3.6 (substitutivity of $\mathcal{E}^*$) and some care is needed in the cases for (I4) and (I5) because they generate additional constraints. The remaining cases are presented in detail below.

(I1) and (P3). Here, $e = v\,v'$, for some v' and where $v = \mathtt{fun}\,f(x : T_{in}) : T_{out} = e''$. We also know that $T = T_{in} \to T_{out}$, and may assume that $f, x \notin dom(\Delta)$. By (I1) and (P5) we get $\exists\Delta(\overline{c}; F; e''v', v/x, f)\downarrow^n$. Now, we know that $\Delta \vdash v\,v'\,\mathcal{E}^*\,e' : T_{out}$ holds. This must have been derived using ($\mathcal{E}^*$-HOWE), and hence we get that

$$\Delta \vdash v\,\mathcal{E}^*\,v_1^* : T_{in} \to T_{out}, \tag{A.1}$$

$\Delta \vdash v'\,\mathcal{E}^*\,v_2^* : T_{in}$ and $\Delta \vdash v_1^*\,v_2^* \cong^\circ e' : T_{out}$ all hold, for some v_1^* and v_2^*. Now, (A.1) could either have been derived using ($\mathcal{E}^*$-FUN) or ($\mathcal{E}^*$-HOWE). Therefore, we perform a case split.

($\mathcal{E}^*$-Fun). We get that $v_1^* = \texttt{fun}\, f\,(x : T_{in}) : T_{out} = e^*$ and $\Delta, f : T_{in} \to T_{out}, x : T_{in} \vdash e'' \cong^\circ e^* : T_{out}$ both hold. By Lemma A.3.3 we get that $\Delta, f : T_{in} \to T_{out}, x : T_{in} \vdash e''\ \mathcal{E}^*\ e^* : T_{out}$, and since $\mathcal{E}^*$ is substitutive (by Lemma A.3.6) we can show that

$$\Delta \vdash e''[v', v / x, f]\ \mathcal{E}^*\ e^*[v_2^*, v_1^* / x, f] : T_{out}.$$

Then, by induction we get that $\exists\Delta(\overline{c}'; F'; e^*[v_2^*, v_1^*/x, f])\downarrow$. Using (I1) and (P5) we have $\exists\Delta(\overline{c}'; F'; v_1^*\ v_2^*)\downarrow$, and since $v_1^*\ v_2^*$ is operationally equivalent to e' we get $\exists\Delta(\overline{c}'; F'; e')\downarrow$, as required.

($\mathcal{E}^*$-Howe). In this case, we get that $\Delta \vdash v\ \widehat{\mathcal{E}^*}\ (\texttt{fun}\, f\,(x : T_{in}) : T_{out} = e^\dagger) : T_{in} \to T_{out}$ and $\Delta \vdash (\texttt{fun}\, f\,(x : T_{in}) : T_{out} = e^\dagger) \cong^\circ v_1^* : T_{in} \to T_{out}$ both hold, for some $e^\dagger$. By applying the compatible refinement rule for functions and Lemma A.3.1 to the $\widehat{\mathcal{E}^*}$ judgement, we can show that

$$\Delta, f : T_{in} \to T_{out}, x : T_{in} \vdash e''\ \mathcal{E}^*\ e^\dagger : T_{out}$$
$$\Delta \vdash v\ \mathcal{E}^*\ (\texttt{fun}\, f\,(x : T_{in}) : T_{out} = e^\dagger) : T_{in} \to T_{out}$$

hold, respectively. Since $\mathcal{E}^*$ is substitutive, we get that

$$\Delta \vdash e''[v, v' / f, x]\ \mathcal{E}^*\ e^\dagger[(\texttt{fun}\, f\,(x : T_{in}) : T_{out} = e^\dagger), v_2^* / f, x] : T_{out}.$$

Then, by using our induction hypothesis we can infer that the termination judgement $\exists\Delta(\overline{c}'; F'; e^\dagger[(\texttt{fun}\, f\,(x : T_{in}) : T_{out} = e^\dagger), v_2^*/f, x])\downarrow$ holds, and by (I1) and (P5) we have

$$\exists\Delta(\overline{c}'; F'; (\texttt{fun}\, f\,(x : T_{in}) : T_{out} = e^\dagger)\ v_2^*)\downarrow.$$

Since $\texttt{fun}\, f\,(x : T_{in}) : T_{out} = e^\dagger$ is operationally equivalent to v_1^*, using an intermediate frame stack we can show that $\exists\Delta(\overline{c}'; F'; v_1^*\ v_2^*)\downarrow$ holds, and because $v_1^*\ v_2^*$ and e' are operationally equivalent, we have $\exists\Delta(\overline{c}'; F'; e')\downarrow$, as required.

(I2). In this case we know that $e = c$ and $T = \texttt{prop}$. Then, by ($\mathcal{E}^*$-Howe) (and the form of the compatible refinement rules for constraints) we know that $\Delta \vdash c\ \widehat{\mathcal{E}^*}\ c' : \texttt{prop}$ and $\Delta \vdash c' \cong^\circ e' : \texttt{prop}$ both hold, for some c. Using the rule for constraint problems from Figure 5.2 we get that $\Delta \vdash \overline{c}\ \&\ c\ \widehat{\mathcal{E}^*}\ \overline{c}'\ \&\ c' : \texttt{prop}$ holds. By applying rule (I4) to the original termination judgement we get that $\exists\Delta(\overline{c}\ \&\ c; F; \mathsf{T})\downarrow^n$ and $\models \exists\Delta(\overline{c}\ \&\ c)$ both hold. Since $\mathcal{E}^*$ is reflexive we know that $\Delta \vdash \mathsf{T}\ \mathcal{E}^*\ \mathsf{T} : \texttt{prop}$ holds, and then, by induction we get that $\exists\Delta(\overline{c}'\ \&\ c'; F'; \mathsf{T})\downarrow$. From Lemma A.3.11 we get that $\models \exists\Delta(\overline{c}'\ \&\ c')$, and then we can use (I4) again to infer that $\exists\Delta(\overline{c}'; F'; c')\downarrow$ holds, and then by operational equivalence we get that $\exists\Delta(\overline{c}'; F'; e')\downarrow$, as required.

This concludes the proof of Lemma A.3.12. □

We can now prove the overall result from this section, that $\cong^\circ$ is compatible in the case of recursive function values.

Lemma A.3.13. *For any two recursive functions* $\texttt{fun}\, f\,(x : T) : T' = e$ *and* $\texttt{fun}\, f\,(x : T) : T' = e'$*:*

if $\quad \Delta \vdash (\texttt{fun}\, f\,(x : T) : T' = e)\ \widehat{\cong^\circ}\ (\texttt{fun}\, f\,(x : T) : T' = e') : T \to T'$
then $\quad \Delta \vdash (\texttt{fun}\, f\,(x : T) : T' = e) \cong^\circ (\texttt{fun}\, f\,(x : T) : T' = e') : T \to T'$.

Proof. We assume that $\Delta \vdash (\texttt{fun}\, f\,(x : T) : T' = e)\ \widehat{\cong^\circ}\ (\texttt{fun}\, f\,(x : T) : T' = e') : T \to T'$ holds—by the compatible refinement rule for functions we know that $\Delta, f : T \to T', x : T \vdash e \cong^\circ e' : T'$ holds. Since $\cong^\circ$ is symmetric we also have $\Delta, f : T \to T', x : T \vdash e' \cong^\circ e : T'$, and if we apply

($\mathcal{E}^*$-FUN) to these we get that $\Delta \vdash (\texttt{fun}\, f\,\texttt{(}x\texttt{:}T\texttt{)}\texttt{:}T' \texttt{=} e)\ \mathcal{E}^*\ (\texttt{fun}\, f\,\texttt{(}x\texttt{:}T\texttt{)}\texttt{:}T' \texttt{=} e') : T \to T'$ and $\Delta \vdash (\texttt{fun}\, f\,\texttt{(}x\texttt{:}T\texttt{)}\texttt{:}T' \texttt{=} e')\ \mathcal{E}^*\ (\texttt{fun}\, f\,\texttt{(}x\texttt{:}T\texttt{)}\texttt{:}T' \texttt{=} e) : T \to T'$ both hold. We pick any $\Delta' \supseteq \Delta$, T'', $\Delta' \vdash F : (T \to T') \to T''$ and $\Delta' \vdash \overline{c} : \texttt{prop}$, and by reflexivity arguments we get that $\Delta' \vdash F\ \widehat{\mathcal{E}^*}\ F : (T \to T') \to T''$ and $\Delta \vdash \overline{c}\ \widehat{\mathcal{E}^*}\ \overline{c} : \texttt{prop}$ both hold. By applying Lemma A.3.12 to both $\mathcal{E}^*$-judgements we get that

$$\exists \Delta'(\overline{c}; F; (\texttt{fun}\, f\,\texttt{(}x\texttt{:}T\texttt{)}\texttt{:}T' \texttt{=} e))\!\downarrow\ \Longleftrightarrow\ \exists \Delta'(\overline{c}; F; (\texttt{fun}\, f\,\texttt{(}x\texttt{:}T\texttt{)}\texttt{:}T' \texttt{=} e'))\!\downarrow.$$

This gives us $\Delta \vdash (\texttt{fun}\, f\,\texttt{(}x\texttt{:}T\texttt{)}\texttt{:}T' \texttt{=} e) \cong^{\circ} (\texttt{fun}\, f\,\texttt{(}x\texttt{:}T\texttt{)}\texttt{:}T' \texttt{=} e') : T \to T'$, as required. □

A.4 Summary of compatibility

We have now covered all of the cases for the derivation of $\Delta \vdash e\ \widehat{\cong^{\circ}}\ e' : T$ so we can now state the following result.

Theorem A.4.1 ($\cong^{\circ}$ is compatible). *For all Γ, e, e', T, if $\Gamma \vdash e\ \widehat{\cong^{\circ}}\ e' : T$ then $\Gamma \vdash e \cong^{\circ} e' : T$.* □

Appendix B

Contextual equivalence of formulae

In this appendix we present a proof of Theorem 5.5.2. In what follows, we will assume that $\emptyset \vdash v_{\mathcal{D}^*} : S_r \to \texttt{prop}$, where $v_{\mathcal{D}^*}$ is the αML encoding of the α-inductive definition $\mathcal{D}^* \in \{\mathcal{D}, \mathcal{D}'\}$. We prove the two directions of the result separately.

B.1 Contextually equivalent formulae have the same semantics

In this section we show that contextual equivalence implies semantic equivalence for encoded formulae. This is one area of the theory where the proofs rely on our underlying nominal sets model of abstract syntax with binders. First, however, we need an auxiliary definition.

Definition B.1.1 (Characteristic expressions). Suppose that the α-tree valuation V is

$$\{x_1 \mapsto [g_1]_\alpha, \dots, x_n \mapsto [g_n]_\alpha\}. \tag{B.1}$$

Then, we write e_V for any *characteristic expression* of V, which is any expression of the form

$$\texttt{let}\, z_1 = [\![g_1]\!]\ \texttt{in} \cdots \texttt{in}\ \texttt{let}\, z_n = [\![g_n]\!]\ \texttt{in}\ x_1 = z_1\ \texttt{\&} \cdots \texttt{\&}\ x_n = z_n$$

where the bound variables $z_1, \dots, z_n$ are pairwise distinct and disjoint from $x_1, \dots, x_n$. ◇

Remark B.1.2 (α-equivalence class representatives). Note that it doesn't matter which representatives of the α-equivalence class we choose, because of the fundamental correctness property of αML (Theorem 5.4.10). This states that the translations of the various trees from a single α-equivalence class are contextually equivalent, and therefore it is irrelevant which one we choose as their operational behaviour will be identical. ◇

Lemma B.1.3 (Typing for characteristic expressions). *Suppose that $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ has the form of* (B.1.1) *above. Then, $\Delta' \vdash e_V : \texttt{prop}$ holds for any $\Delta' \supseteq \Delta$ such that $\Delta' \vdash FN(g_1, \dots, g_n)$.*

Proof. The proof is standard, using the typing rules from Figure 3.2 and Lemma 5.4.4. The assumption that $\Delta' \vdash FN(g_1, \dots, g_n)$ is needed to ensure that the variables corresponding to translated names are always assigned the correct type. □

We now consider the behaviour of characteristic expressions when they are evaluated. There is a potential problem here: recall that the definition of a characteristic expression e_V of a valuation $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ involves translated ground trees $[\![g_i]\!]$. A free name n of the tree g_i corresponds to a free variable $\mathcal{V}(n)$ of $[\![g_i]\!]$, and hence of e_V. Since the bijection $\mathcal{V}(-)$ that maps

names to variables is fixed, it is possible that one of the free variables $\mathcal{V}(n)$ could clash with a variable from $dom(\Delta)$. This is problematic because the variables which are used to represent the ground trees $[\![g_i]\!]$ are not related to the variables which appear $dom(\Delta)$. In the following lemmas we will assume that $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ of the form (B.1) is such that following property holds.

$$\{\mathcal{V}(n) \mid n \in FN(g_1, \ldots, g_n)\} \cap \{x_1, \ldots, x_n\} = \emptyset \tag{B.2}$$

In the proof of the main theorem in this section (Theorem B.1.6) below we will use an argument based on equivariance to show that this problem can be avoided.

Lemma B.1.4 (Evaluation of characteristic expressions). *Suppose that* $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ *is as in* (B.1.1) *and has the property* (B.2)*. Pick arbitrary* Δ'*,* F *and* T *such that* $\Delta' \supseteq \Delta$*,* $\Delta' \vdash FN(g_1, \ldots, g_n)$ *and* $\Delta' \vdash F : \mathtt{prop} \to T$ *all hold. Then, there exist* η_V *and* $\overline{c}_V$ *such that*

$$\exists \Delta'(\mathsf{T}; F; e_V) \longrightarrow \cdots \longrightarrow \exists \Delta', \eta_V(\overline{c}_V; F; \mathsf{T}) \tag{B.3}$$

and $\models \exists \Delta', \eta_V(\overline{c}_V)$ *both hold. Furthermore, for any* $V^* \in \alpha\text{-}Tree_\Sigma(\Delta', \eta_V)$*, if* $V^* \models \overline{c}_V$ *then there exists a permutation* π^* *such that* $V^*(x) = \pi^* \cdot V(x)$ *for all* $x \in dom(\Delta)$*.*

Proof. We know the form of the expression e_V from Definition B.1.1. Since the evaluation of the ground trees $[\![g_i]\!]$ only produces freshness constraints it follows that evaluating the ground trees will succeed: now suppose that evaluating $[\![g_i]\!]$ produces η_i, $\overline{c}_i$ and v_i. Therefore we get that

$$\exists \Delta'(\mathsf{T}; F; e_V) \longrightarrow \cdots \longrightarrow \exists \Delta', \eta_1, \ldots, \eta_n(\overline{c}_1 \;\&\; \cdots \;\&\; \overline{c}_n; F; x_1 = v_1 \;\&\; \cdots \;\&\; x_n = v_n) \tag{B.4}$$

Now, since we are assuming that V has the property (B.2) it follows that the assignments to the variables $x_1, \ldots, x_n$ in the second configuration of (B.4) cannot conflict with the freshness constraints $\overline{c}_1 \;\&\; \cdots \;\&\; \overline{c}_n$. Therefore from (B.4) it follows that (B.3) holds, where $\eta_V = \eta_1, \ldots, \eta_n$ and $\overline{c}_V = \overline{c}_1 \;\&\; \cdots \;\&\; \overline{c}_n \;\&\; x_1 = v_1 \;\&\; \cdots \;\&\; x_n = v_n$. Because the constraints in $\overline{c}_1, \ldots, \overline{c}_n$ are all freshnesses and because the variables $x_1, \ldots, x_n$ do not appear elsewhere, it follows that $\models \exists \Delta', \eta_V(\overline{c}_V)$.

Now, suppose that $V^* \in \alpha\text{-}Tree_\Sigma(\Delta', \eta_V)$ is such that $V^* \models \overline{c}_V$. By the semantics of constraints it follows that $V^*(x) = [\![v_i]\!]_{V^*}$ holds for all $i \in \{1, \ldots, n\}$. Then, by Lemma 5.4.8 we get that there exists a permutation π^* such that $(\pi^* \cdot g_i) \in [\![v_i]\!]_{V^*}$ for all $i \in \{1, \ldots, n\}$. Finally, since $V(x_i) = [g_i]_\alpha$ by its definition from (B.1.1), it follows that $V^*(x_i) = \pi^* \cdot V(x_i)$ for all $i \in \{1, \ldots, n\}$, as required. ☐

The previous lemma formalised the sense in which a characteristic expression e_V represents the α-tree valuation V. We now prove the central lemma of this proof, in which we show that the expression $e_V \;\&\; \varphi[v_\mathcal{D}/r]$ terminates in the empty context iff $([\![\mathcal{D}]\!], V) \models \varphi$.

Lemma B.1.5. *Suppose that* $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ *has the form of* (B.1.1) *above, and satisfies the property* (B.2)*. We pick an arbitrary* Δ' *such that* $\Delta' \supseteq \Delta$ *and* $\Delta' \vdash FN(g_1, \ldots, g_n)$ *both hold. Let* $\mathcal{D}$ *be an arbitrary* α*-inductive definition (in standard form) and let* φ *be a schematic formula such that* $\Delta' \vdash \varphi[v_\mathcal{D}/r] : \mathtt{prop}$ *holds. Then it is the case that*

$$\exists \Delta'(\mathsf{T}; \mathtt{Id}; e_V \;\&\; \varphi[v_\mathcal{D}/r])\!\downarrow \iff ([\![\mathcal{D}]\!], V) \models \varphi. \tag{B.5}$$

Proof. By Lemma B.1.4 we get that

$$\exists \Delta'(\mathsf{T}; \mathtt{Id}; e_V \;\&\; \varphi[v_\mathcal{D}/r]) \longrightarrow \cdots \longrightarrow \exists \Delta', \eta_V(\overline{c}_V; F; \varphi[v_\mathcal{D}/r]) \tag{B.6}$$

and $\models \exists \Delta', \eta_V(\overline{c}_V)$ both hold. Furthermore, for any $V^* \in \alpha\text{-}Tree_\Sigma(\Delta', \eta_V)$, if $V^* \models \overline{c}_V$ then there exists a permutation π^* such that $V^*(x) = \pi^* \cdot V(x)$ for all $x \in dom(\Delta)$. Now we prove the two directions of (B.5) separately.

$\exists\Delta'(\mathtt{T};\mathtt{Id};e_V \mathbin{\&} \varphi[v_{\mathcal{D}}/r])\!\downarrow \implies ([\![\mathcal{D}]\!], V) \models \varphi$.
From $\exists\Delta'(\mathtt{T};\mathtt{Id};e_V \mathbin{\&} \varphi[v_{\mathcal{D}}/r])$ and (B.6) we get that

$$\exists\Delta', \eta_V(\overline{c}_V; F; \varphi[v_{\mathcal{D}}/r])\!\downarrow \tag{B.7}$$

holds, from which it follows that there exist Δ_φ and $\overline{c}_\varphi$ such that

$$\exists\Delta_\varphi(\overline{c}_\varphi) \in solns_{\mathcal{D}}((\Delta', \eta_V), \overline{c}_V, \emptyset, \varphi) \tag{B.8}$$
$$V^* \models \exists\Delta_\varphi(\overline{c}_\varphi) \tag{B.9}$$

both hold, for some $V^* \in \alpha\text{-}Tree_\Sigma(\Delta', \eta_V)$. From (B.9) it follows that $V^* \models \overline{c}_V$ and hence that

$$V^*(x) = \pi^* \cdot V(x) \tag{B.10}$$

holds for all $x \in dom(\Delta)$ and for some permutation π^*. By (B.8), (B.9) and Logical Soundness (Theorem 4.3.2) we get that $([\![\mathcal{D}]\!], V^*) \models \overline{c}_V \mathbin{\&} \varphi$ holds, which we simplify to $([\![\mathcal{D}]\!], V^*) \models \varphi$. By this and (B.10) we get that $([\![\mathcal{D}]\!], \pi^* \cdot V) \models \varphi$ holds. Finally, by equivariance (Lemma 2.5.6 and Lemma 2.5.7) it follows that $([\![\mathcal{D}]\!], V) \models \varphi$ holds, as required.

$([\![\mathcal{D}]\!], V) \models \varphi \implies \exists\Delta'(\mathtt{T};\mathtt{Id};e_V \mathbin{\&} \varphi[v_{\mathcal{D}}/r])$.
From $([\![\mathcal{D}]\!], V) \models \varphi$, (B.6) and $\models \exists\Delta', \eta_V(\overline{c}_V)$ we know that there exist a permutation π^* and a valuation $V^* \in \alpha\text{-}Tree_\Sigma(\Delta', \eta_V)$ such that

$$V^* \models \overline{c}_V \tag{B.11}$$
$$V^*(x) = \pi^* \cdot V(x) \tag{B.12}$$

hold, for all $x \in dom(\Delta)$. Now, using the results about equivariance from Lemma 2.5.6 and Lemma 2.5.7, along with our initial assumption that $([\![\mathcal{D}]\!], V) \models \varphi$ we conclude that $([\![\mathcal{D}]\!], \pi^* \cdot V) \models \varphi$. From this, (B.11) and (B.12) we get that $V^* \models \overline{c}_V \mathbin{\&} \varphi$ holds. Using this fact, along with the Logical Completeness result from Theorem 4.3.2, we get that there exist Δ_φ and $\overline{c}_\varphi$ such that

$$\exists\Delta_\varphi(\overline{c}_\varphi) \in solns_{\mathcal{D}}((\Delta', \eta_V), \overline{c}_V, \emptyset, \overline{c}) \tag{B.13}$$
$$V^* \models \exists\Delta_\varphi(\overline{c}_\varphi) \tag{B.14}$$

both hold. Finally, by (B.6), (B.13) and (B.14) we get that $\exists\Delta'(\mathtt{T};\mathtt{Id};e_V \mathbin{\&} \varphi[v_{\mathcal{D}}/r])\!\downarrow$, as required.

This completes the proof of Lemma B.1.5. □

The proof of Lemma B.1.5 relies on the fact that ground trees can be represented in αML in a way that respects α-equivalence. More specifically, it relies on details of the underlying semantics of schematic formulae in terms of α-equivalence classes of ground trees (which form a nominal set). Arguments based on equivariance are typical in the world of nominal sets and nominal logic (Pitts, 2003, 2006) but are largely absent from this dissertation. Although the proofs of some important meta-theoretic results about the operational behaviour of αML programs (such as Theorem 5.4.10 and Theorem 5.5.2) rely on these concepts, the statements of the results themselves do not. Indeed, we see it as a positive thing that the details of the underlying mathematical model are hidden from view to such a degree.

We now prove the main result of this section, that if two formulae are contextually equivalent then they have the same semantics. Again, we use equivariance to argue that it is sufficient to consider valuations which have the property (B.2).

Theorem B.1.6. *For all $\mathcal{D}$, $\mathcal{D}'$, Δ, φ and φ', if $\Delta \vdash \varphi[v_{\mathcal{D}}/r] \cong \varphi'[v_{\mathcal{D}'}/r]:\texttt{prop}$ then $\mathcal{D}, \mathcal{D}' \models \forall\Delta.\, \varphi \equiv \varphi'$.*

Proof. We assume that $\Delta \vdash \varphi[v_{\mathcal{D}}/r] \cong \varphi'[v_{\mathcal{D}'}/r]:\texttt{prop}$ and pick an arbitrary valuation $V \in \alpha\text{-}Tree_{\Sigma}(\Delta)$: we must show that $(\llbracket\mathcal{D}\rrbracket, V) \models \varphi \iff (\llbracket\mathcal{D}'\rrbracket, V) \models \varphi'$. By equivariance this is equivalent to $(\llbracket\mathcal{D}\rrbracket, \pi \cdot V) \models \varphi \iff (\llbracket\mathcal{D}'\rrbracket, \pi \cdot V) \models \varphi'$ for any permutation π. Now, if V does not have the disjointness property (B.2) then we can always find a suitable permutation π to produce a valuation $\pi \cdot V$ which does have that property. Therefore it is sufficient to consider valuations which satisfy the property (B.2). Now, we pick an arbitrary type environment Δ' such that $\Delta' \supseteq \Delta$ and $\Delta' \vdash FN(g_1, \ldots, g_n)$ both hold. Then, since $\Delta \vdash \varphi[v_{\mathcal{D}}/r] \cong \varphi'[v_{\mathcal{D}'}/r]:\texttt{prop}$ we know in particular that

$$\exists\Delta, \eta(\texttt{T};\texttt{Id};\varphi[v_{\mathcal{D}}/r]\ \texttt{\&}\ e_V)\!\downarrow \quad\iff\quad \exists\Delta, \eta(\texttt{T};\texttt{Id};\varphi'[v_{\mathcal{D}'}/r]\ \texttt{\&}\ e_V)\!\downarrow.$$

Then, by Lemma B.1.5 it follows that $(\llbracket\mathcal{D}\rrbracket, V) \models \varphi \iff (\llbracket\mathcal{D}'\rrbracket, V) \models \varphi'$. Thus we have shown that $\mathcal{D}, \mathcal{D}' \models \forall\Delta.\, \varphi \equiv \varphi'$ holds, as required. ☐

B.2 Formulae with the same semantics are contextually equivalent

In this section we aim to prove that if two formulae have the same semantics then they have identical termination behaviour in any αML evaluation context. This direction of the proof is quite delicate, and relies on certain properties of the αML reduction relation. We first prove the following preparatory lemma, which demonstrates that two configurations which are almost identical, in that their constraints are different but have a satisfying valuation in common, may take similar reduction steps. This is a general property of all expressions, not just of the expressions corresponding to formulae which concern us in this section.

Lemma B.2.1. *Suppose that $\emptyset \vdash \exists\Delta, \Delta_1(\overline{c}_1; F; e) : T$ and $\emptyset \vdash \exists\Delta, \Delta_2(\overline{c}_2; F; e) : T$ both hold, and that*

$$\exists\Delta, \Delta_1(\overline{c}_1; F; e) \longrightarrow \exists\Delta, \Delta_1, \Delta'(\overline{c}_1\ \texttt{\&}\ \overline{c}'; F'; e') \tag{B.15}$$

holds for some $\Delta', \overline{c}', F', e'$. Then, for any $V \in \alpha\text{-}Tree_{\Sigma}(\Delta)$, if $V \models \exists\Delta_1, \Delta'(\overline{c}_1\ \texttt{\&}\ \overline{c}')$ and $V \models \exists\Delta_2(\overline{c}_2)$ both hold, then $\exists\Delta, \Delta_2(\overline{c}_2; F; e) \longrightarrow \exists\Delta, \Delta_2, \Delta'(\overline{c}_2\ \texttt{\&}\ \overline{c}'; F'; e')$ and $V \models \exists\Delta_2, \Delta'(\overline{c}_2\ \texttt{\&}\ \overline{c}')$ both hold.

Proof. We assume that (B.15) holds, and that $V \models \exists\Delta_1, \Delta'(\overline{c}_1\ \texttt{\&}\ \overline{c}')$ and $V \models \exists\Delta_2(\overline{c}_2)$ both hold, for $V \in \alpha\text{-}Tree_{\Sigma}(\Delta')$. We proceed by a case split on the impure reduction rule used to derive (B.15).

(I1). Assume that (B.15) is derived using (I1). Then, it follows that $\Delta' = \emptyset$ and $\overline{c}' = \emptyset$, and also that$\langle F, e\rangle \to_P \langle F', e'\rangle$. Using this pure reduction, we may deduce (using (I1) again) that

$$\exists\Delta, \Delta_2(\overline{c}_2; F; e) \longrightarrow \exists\Delta, \Delta_2(\overline{c}_2; F'; e')$$

i.e. that $\exists\Delta, \Delta_2(\overline{c}_2; F; e) \longrightarrow \exists\Delta, \Delta_2, \Delta'(\overline{c}_2\ \texttt{\&}\ \overline{c}'; F; e)$ holds. By assumption we get that $V \models \exists\Delta_2, \Delta'(\overline{c}_2\ \texttt{\&}\ \overline{c}')$ also holds, as required.

(I2). If (I2) is used to derive (B.15), we know that $e = x.\,i$, for some variable x such that $(x : E_1 * \cdots * E_n) \in \Delta, \Delta_1$ and $(x : E_1 * \cdots * E_n) \in \Delta, \Delta_2$. Then, $\Delta' = \{x_1 : E_1, \ldots, x_n : E_n\}$ where the variables $x_1, \ldots, x_n$ are mutually distinct and do not appear in $dom(\Delta, \Delta_1, \Delta_2, \Delta')$. Furthermore we have $\overline{c}' = x\,\texttt{=}\,(x_1, \ldots, x_n)$, $F' = F$ and $e' = x_i$. Using (I2) we can show that $\exists\Delta, \Delta_2(\overline{c}_2; F; x.\,i) \longrightarrow \exists\Delta, \Delta_2, \Delta'(\overline{c}_2\ \texttt{\&}\ \overline{c}'; F; x_i)$ holds, i.e. we have shown that

$$\exists\Delta, \Delta_2(\overline{c}_2; F; e) \longrightarrow \exists\Delta, \Delta_2, \Delta'(\overline{c}_2\ \texttt{\&}\ \overline{c}'; F'; e').$$

Finally, since the new variables $x_1, \ldots, x_n$ are unconstrained we have that $V \models \exists\Delta_2, \Delta'(\overline{c}_2\ \texttt{\&}\ \overline{c}')$, as required.

(I3). When we use (I3) to derive (B.15), we know that $e = \texttt{case}\ x\ \texttt{of}\ K_1\, x_1 \,\texttt{->}\, e_1 \mid \cdots \mid K_n\, x_n \,\texttt{->}\, e_n$, for some variable x where $(x\!:\!S) \in \Delta, \Delta_1$, $(x\!:\!S) \in \Delta, \Delta_2$ and $\texttt{datatype}\ S =_{\Sigma} K_1\ \texttt{of}\ E_1 \mid \cdots \mid K_n\ \texttt{of}\ E_n$ all hold, for some S. Then, by matching against rule (I3) we also get that $\Delta' = \{x_i\!:\!E_i\}$, $\overline{c}' = x = K_i\, x_i$, $F' = F$ and $e' = e_i$, for some $i \in \{1, \ldots, n\}$ and where $x_i \notin dom(\Delta, \Delta_1, \Delta_2)$. By assumption we also know that $V \models \exists\Delta_1, \Delta'(\overline{c}\ \&\ \overline{c}')$. From this we may deduce that $V \models \exists\Delta_2, \Delta'(\overline{c}\ \&\ \overline{c}')$, and hence that $\models \exists\Delta, \Delta_2, \Delta'(\overline{c}\ \&\ \overline{c}')$. By (I3) we get

$$\exists\Delta, \Delta_2(\overline{c}_2; F; \texttt{case}\ x\ \texttt{of}\ K_1\, x_1 \,\texttt{->}\, e_1 \mid \cdots \mid K_n\, x_n \,\texttt{->}\, e_n) \longrightarrow \exists\Delta, \Delta_2, x_i\!:\!E_i(\overline{c}_2\ \&\ x = K_i\, x_i; F; e_i),$$

i.e. that $\exists\Delta, \Delta_2(\overline{c}_2; F; e) \longrightarrow \exists\Delta, \Delta_2, \Delta'(\overline{c}_2\ \&\ \overline{c}'; F'; e')$ and $V \models \exists\Delta_2, \Delta'(\overline{c}\ \&\ \overline{c}')$ both hold, as required.

(I4). In this case, (B.15) is derived using (I4), and we may infer that $e = c$, $\Delta' = \emptyset$, $\overline{c}' = c$, $F' = F$, $e' = \mathsf{T}$ and $\models \exists\Delta, \Delta_1(\overline{c}_1\ \&\ c)$. Now, since $V \models \exists\Delta_1, \Delta'(\overline{c}_1\ \&\ \overline{c}')$ and $V \models \exists\Delta_2(\overline{c}_2)$ it follows that $V \models \exists\Delta_2(\overline{c}_2\ \&\ c)$, and hence that $\models \exists\Delta_2, \Delta'(\overline{c}_2\ \&\ \overline{c}')$. Then, using rule (I4) we can show that $\exists\Delta, \Delta_2(\overline{c}_2; F; c) \longrightarrow \exists\Delta, \Delta_2(\overline{c}_2\ \&\ \{c\}; F; \mathsf{T})$ holds, from which we may infer that $\exists\Delta, \Delta_2(\overline{c}_2; F; e) \longrightarrow \exists\Delta, \Delta_2, \Delta'(\overline{c}_2\ \&\ \overline{c}'; F'; e')$ and $V \models \exists\Delta_2, \Delta'(\overline{c}_2\ \&\ \overline{c}')$ both hold, as required.

(I5). If (B.15) is derived with rule (I5) we get that $e = \exists x\!:\!E.\, e^*$, where we may assume (by α-conversion) that $x \notin dom(\Delta, \Delta_1, \Delta_2)$. We also assume that $\Delta' = \{x\!:\!E\}$, $\overline{c}' = \emptyset$, $F' = F$ and $e' = e^*$. Hence, by (I5) we get that $\exists\Delta, \Delta_2(\overline{c}_2; F; \exists x\!:\!E.\, e^*) \longrightarrow \exists\Delta, \Delta_2, x\!:\!E(\overline{c}_2; F; e^*)$ holds, from which we can show that $\exists\Delta, \Delta_2(\overline{c}_2; F; e) \longrightarrow \exists\Delta, \Delta_2, \Delta'(\overline{c}_2\ \&\ \overline{c}'; F'; e')$. It also follows (by weakening) that $V \models \exists\Delta_2, \Delta'(\overline{c}_2\ \&\ \overline{c}')$, as required.

This completes the proof of Lemma B.2.1. □

We can now present a proof of the main result in this section, which is that semantically-equivalent formulae are always contextually equivalent in the meta-language.

Theorem B.2.2. *For all $\mathcal{D}, \mathcal{D}', \Delta, \varphi, \varphi'$, if $\mathcal{D}, \mathcal{D}' \models \forall\Delta.\, \varphi \equiv \varphi'$ then $\Delta \vdash \varphi[v_{\mathcal{D}}/r] \cong \varphi'[v_{\mathcal{D}'}/r] : \texttt{prop}$.*

Proof. We assume that $\mathcal{D}, \mathcal{D}' \models \forall\Delta.\, \varphi \equiv \varphi'$, and pick arbitrary Δ', $\overline{c}$, F and T such that $\Delta' \supseteq \Delta$, $\Delta' \vdash \overline{c} : \texttt{prop}$ and $\Delta' \vdash F : \texttt{prop} \to T$. By weakening our assumption we can show that $\mathcal{D}, \mathcal{D}' \models \forall\Delta'.\, \overline{c}\ \&\ \varphi \equiv \overline{c}\ \&\ \varphi'$.

Now, we will assume that $\exists\Delta'(\overline{c}; F; \varphi[v_{\mathcal{D}}/r])\!\downarrow$. By the definition of success, and type safety, and the fact that Δ' and $\overline{c}$ only get larger across $\longrightarrow$-transitions, it follows that

$$\exists\Delta'(\overline{c}; F; \varphi[v_{\mathcal{D}}/r]) \longrightarrow \cdots \longrightarrow \exists\Delta', \Delta_1(\overline{c}\ \&\ \overline{c}_1; F; \mathsf{T}) \tag{B.16}$$

$$\exists\Delta', \Delta_1(\overline{c}\ \&\ \overline{c}_1; F; \mathsf{T}) \longrightarrow \cdots \longrightarrow \exists\Delta', \Delta_1, \Delta_2(\overline{c}\ \&\ \overline{c}_1\ \&\ \overline{c}_2; \texttt{Id}; v) \tag{B.17}$$

$$\models \exists\Delta', \Delta_1, \Delta_2(\overline{c}\ \&\ \overline{c}_1\ \&\ \overline{c}_2) \tag{B.18}$$

all hold, for some $\Delta_1, \Delta_2, \overline{c}_1, \overline{c}_2$. From (B.18) we know that there exists $V \in \alpha\text{-}Tree_{\Sigma}(\Delta')$ such that $V \models \exists\Delta_1, \Delta_2(\overline{c}\ \&\ \overline{c}_1\ \&\ \overline{c}_2)$. By Logical Soundness (Theorem 4.3.2) and (B.16) we can show that $([\![\mathcal{D}]\!], V) \models \varphi\ \&\ \overline{c}$, and by our initial assumption ($\mathcal{D}, \mathcal{D}' \models \forall\Delta.\, \varphi \equiv \varphi'$) it follows that $([\![\mathcal{D}']\!], V) \models \varphi'\ \&\ \overline{c}$. Then, by Logical Completeness (Theorem 4.3.2) we know that there exists $\exists\Delta_3(\overline{c}_3) \in solns_{\mathcal{D}'}(\Delta', \overline{c}, \emptyset, \varphi')$ such that $V \models \exists\Delta_3(\overline{c}_3)$, i.e. such that $\exists\Delta'(\overline{c}; \texttt{Id}; \varphi'[v_{\mathcal{D}'}/r]) \longrightarrow \cdots \longrightarrow \exists\Delta', \Delta_3(\overline{c}\ \&\ \overline{c}_3; \texttt{Id}; \mathsf{T})$ and $\models \exists\Delta', \Delta_3(\overline{c}\ \&\ \overline{c}_3)$. Since we can add extra frames at the bottom of the stack without affecting the validity of the $\longrightarrow$-judgements, we get that

$$\exists\Delta'(\overline{c}; F; \varphi'[v_{\mathcal{D}'}/r]) \longrightarrow \cdots \longrightarrow \exists\Delta', \Delta_3(\overline{c}\ \&\ \overline{c}_3; F; \mathsf{T}) \tag{B.19}$$

$$\models \exists\Delta', \Delta_3(\overline{c}\ \&\ \overline{c}_3) \tag{B.20}$$

both hold. By applying Lemma B.2.1 to (B.17), (B.18), (B.19) and (B.20) we get that

$$\exists\Delta', \Delta_3(\overline{c}\ \texttt{\&}\ \overline{c}_3; F; \mathsf{T}) \longrightarrow \cdots \longrightarrow \exists\Delta', \Delta_3, \Delta_2(\overline{c}\ \texttt{\&}\ \overline{c}_3\ \texttt{\&}\ \overline{c}_2; \texttt{Id}; v) \tag{B.21}$$

$$V \models \exists\Delta_3, \Delta_2(\overline{c}\ \texttt{\&}\ \overline{c}_3\ \texttt{\&}\ \overline{c}_2) \tag{B.22}$$

both hold, and by (B.19), (B.21) and (B.22) we get that $\exists\Delta'(\overline{c}; F; \varphi'[v_{\mathcal{D}'}/r])\downarrow$.

By a similar argument we can show that $\exists\Delta'(\overline{c}; F; \varphi'[v_{\mathcal{D}'}/r])\downarrow$ implies $\exists\Delta'(\overline{c}; F; \varphi[v_{\mathcal{D}}/r])\downarrow$. Therefore we have that $\exists\Delta'(\overline{c}; F; \varphi[v_{\mathcal{D}}/r])\downarrow \iff \exists\Delta'(\overline{c}; F; \varphi'[v_{\mathcal{D}'}/r])\downarrow$. Hence, we have shown that $\Delta \vdash \varphi[v_{\mathcal{D}}/r] \cong \varphi'[v_{\mathcal{D}'}/r] : \texttt{prop}$ holds, as required. $\square$

Appendix C

Implementation details

This appendix comprises a detailed description of the αML runtime. We present the custom bytecode which is the target language for our compiler and informally describe its semantics. We then define a compilation function from core αML to bytecode machine instructions.

C.1 The bytecode machine

The αML bytecode machine is loosely based on Landin's SECD machine (Landin, 1964). The code for implementing anonymous functions and application is an extension of Andrej Bauer's implementation of Mini-ML (Bauer, 2008), with many more features been added to implement the advanced features of αML.

We begin by defining a language of *machine values*, *mv*. These are given by the following grammar.

$$\begin{array}{rcl} mv & ::= & \texttt{MVar}(n) \mid \texttt{MEvar}(n) \mid \texttt{MUnit} \mid \texttt{MYes} \mid \texttt{MData}(K\,, mv) \mid \\ & & \texttt{MTuple}(mv_1, \ldots, mv_n) \mid \texttt{MAbs}(mv\,, mv') \mid \texttt{MClosure}(I\,, E). \end{array}$$

The close similarity with the grammar of αML values v from Figure 3.1 is no coincidence. The most interesting points are the integer arguments of the first two constructors and the arguments of `MClosure`, which will be discussed below. In $\texttt{MVar}(n)$, the integer n is a de Bruijn index into an environment which stores the assignments to the bound value identifiers, and in $\texttt{MEvar}(n)$, the integer is a tag which identifies a particular existential variable.

Definition C.1.1 (Bytecode machine states). States of the αML bytecode machine take the form $\langle I, S, \vec{E}, \mathbb{V}, \overline{c} \rangle$ where:

- ***I*** is a finite list of *instructions*, which are discussed below. These are the code remaining to be executed in this branch of the computation.
- ***S*** is a *stack*, which is a finite list of machine values. This stores the values being operated on by the bytecode machine.
- $\vec{E}$ is a finite list of *environments* E, which are finite lists of machine values. This represents the statically-scoped closing environments of the functions in the call stack.
- $\mathbb{V}$ is a data structure which stores information on the *existential variables* generated so far. It is implemented using the `Vec` library (de Alfaro, 2008) which implements functional arrays using balanced binary trees. Lookup and insertion operations take amortised logarithmic time. If the existential variable $\texttt{MEvar}(i)$ has been dynamically generated, then the array cell $\mathbb{V}[i]$ holds its type and (possibly) a machine value which is its current instantiation.

Instruction	Description
`IVar`(n)	Pushes `MVar`(n) onto the stack.
`IEvar`(n)	Pushes `MEvar`(n) onto the stack.
`IUnit`	Pushes `MUnit` onto the stack.
`IYes`	Pushes `MYes` onto the stack.
`IData`(K)	Replaces mv at the top of the stack with `MData`(K, mv).
`ITuple`(n)	Replaces the top n values $mv_1, \ldots, mv_n$ at the top of the stack with `MTuple`$(mv_1, \ldots, mv_n)$.
`IAbs`	Replaces mv and mv' at the top of the stack with `MAbs`(mv, mv').
`IClosure`(I)	Grabs the current environment E and pushes the closure `MClosure`(I, E) onto the stack.
`IRecClosure`(I)	Grabs the current environment E and pushes the recursive closure `MClosure`(I, E) onto the stack.
`ICall`	Performs a function call with a closure `MClosure`(I, E) and argument mv popped from the stack.
`IPopEnv`	Pops the top environment from the stack of environments and discards it.
`IDiscardStkTop`	Pops the top element from the stack and discards it.
`IDiscardEnvTop`	Pops the top element from the environment and discards it.
`IStkToEnv`	Pops the top element from the stack and pushes it onto the environment.
`IBranch`$(\vec{I})$	Causes a non-deterministic choice between the branches $\vec{I}$.
`IFail`	Causes the current branch of computation to fail immediately.
`IProj`(n)	Performs a rigid or flexible projection, depending on whether `MTuple`$(mv_1, \ldots, mv_k)$ or `MEvar`(j) is on top of the stack.
`ICase`$(\vec{K}, \vec{I})$	Evaluates a rigid or flexible `case` expression on the clauses $\vec{K}, \vec{I}$, depending on whether `MData`(K, mv) or `MEvar`(j) is on top of the stack.
`IEcon`	Pops mv and mv' off the stack and tries to solve the equality constraint $mv = mv'$.
`IFcon`	Like `IEcon`, but solves the freshness constraint $mv \# mv'$ instead.
`IExists`(E)	Pushes a new existential variable `MEvar`(n) onto the stack.
`IFresh`(N)	Has the same effect as `IExists`(N) but also creates freshness constraints `MEvar`(n) # `MEvar`(k) for all $k < n$.
`IDistinct`(n)	Pops $mv_1, \ldots, mv_n$ off the stack and processes all freshness constraints $mv_i \# mv_j$ where $1 \leq i < j \leq n$.

Figure C.1: Informal semantics of bytecode instructions

- $\bar{c}$ carries additional information on the current constraints (on top of the substitution information carried in $\mathbb{V}$). ◇

We can now discuss the closure machine value `MClosure`(I, E). The first argument is the list of instructions corresponding to the body of the function, and the second is the environment of the closure, in which the function should be evaluated (an instance of static scoping).

Figure C.1 presents a list of the different instructions and an informal description of their semantics. There is a large number of instructions: for reasons of space we will not present a formal semantics for the bytecode machine instruction set. We will, however, discuss the execution of certain instructions from Figure C.1 in more detail.

The instruction `IClosure`(I) creates a closure for the function whose body compiles to the instruction list I. It does this by capturing the current environment of bindings for the bound value identifiers (the environment E that is the head of the environment stack $\vec{E}$) and pushing the machine value `MClosure`(I, E) onto the stack. `IRecClosure`(I) is identical to `IClosure`(I) except that it creates a recursive closure by extending the environment E with a circular reference to the closure itself. This implements recursion cheaply.

Function calls are implemented using the `ICall` instruction. This pops a machine value mv and a closure `MClosure`(I, E) from the stack and carries out the function call by appending I to the front of the list of instructions waiting to be executed. The environment for this function call is $mv :: E$, created using the argument mv and the statically-scoped closure environment E. This is pushed onto the stack of environments $\vec{E}$. When a function call returns, the instruction `IPopEnv` should be executed, which discards the top environment from $\vec{E}$. This ensures that the calling function continues its execution in the correct environment.

The instructions `IBranch` and `IFail` deal with branching and finite failure. The transition relation for bytecode machine states is non-deterministic and `IBranch`$(\vec{I})$ is one instruction that causes branching: it produces a set of new states where each state executes one list of instructions I from $\vec{I}$. The `IFail` instruction transitions to the empty set of states, which corresponds to finite failure for that particular branch of the computation.

The `IProj` and `ICase` instructions are particularly interesting because they behave differently depending on the machine value at the top of the stack. These instructions perform the standard projection and pattern-matching operations for a functional programming langauge if a tuple value (`MTuple`$(mv_1, \ldots, mv_n)$) or data value (`MData`(K, mv)) is at the top of the stack, respectively. However, the other possibility is that an existential variable (`MEvar`(j)) could be at the top of the stack. In this case, these instructions perform a flexible projection or `case` expression, respectively. Implementing this second behaviour requires some bytecode machine instructions to be generated at runtime, to generate the existential variables and constraints needed to implement the transition rules (I4) and (I5) from Figure 3.4. The instructions involved here are `IExists`, `IEcon` and `IFcon`.

The `IExists`(E) instruction looks into the $\mathbb{V}$ data structure to find integer index n of the next available existential variable and updates $\mathbb{V}$ to reflect that existential variable number n has been initialised with type E (the type information is used by the constraint solver). The existential variable itself is returned by pushing `MEvar`(n) onto the stack. The instructions `IEcon` and `IFcon` both remove the top two stack elements (mv and mv', say) and attempt to solve the appropriate constraint (either $mv = mv'$ or $mv \# mv'$) in conjunction with the existing constraints stored in the state. These instructions call upon the constraint transformation procedure outlined in Chapter 6.

As mentioned in Section 7.2, the two instructions at the bottom of Figure C.1 are not strictly necessary to implement the core αML language but provide additional functionality.

C.2 Compiling αML expressions

We conclude our description of the αML bytecode machine with a discussion of compiling expressions in the extended αML language.

We will treat the αML language from Chapter 3 and the extensions described in Section 7.2 which are not implemented by translation into core αML. This requires us to convert value identifiers x into the integers which serve as de Bruijn indices within the bytecode machine. For this we use *symbol tables*, st, which are simply (finite) lists of value identifiers. We use the list consing notation $x :: st$ to add a value identifier x onto the head of st, and write $st(x)$ for the index of the first occurrence of x in st. Since we cons a value identifier onto the head of the

$$
\begin{aligned}
\mathcal{C}(x\,,st) &\triangleq \mathtt{MVar}(i) \quad \text{where } st(x) = i & \mathcal{C}(\mathtt{()}\,,st) &\triangleq [\mathtt{IUnit}] \\
\mathcal{C}(\mathtt{?x}\,,st) &\triangleq \mathtt{MEvar}(j) \quad \text{where } j = \mathcal{N}(\mathtt{?x}) & \mathcal{C}(\mathtt{yes}\,,st) &\triangleq [\mathtt{IYes}] \\
\mathcal{C}(\mathtt{<}e\mathtt{>}e'\,,st) &\triangleq \mathcal{C}(e\,,st)\,@\,\mathcal{C}(e'\,,st)\,@\,[\mathtt{IAbs}] & \mathcal{C}(\mathtt{no}\;T\,,st) &\triangleq [\mathtt{IFail}] \\
\mathcal{C}(e.i\,,st) &\triangleq \mathcal{C}(e\,,st)\,@\,[\mathtt{IProj}(i)] & \mathcal{C}(\mathtt{some}\;E\,,st) &\triangleq [\mathtt{IExists}(E)] \\
\mathcal{C}(K\,e\,,st) &\triangleq \mathcal{C}(e\,,st)\,@\,[\mathtt{IData}(K)] & \mathcal{C}(\mathtt{fresh}\;N\,,st) &\triangleq [\mathtt{IFresh}(N)]
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}(\mathtt{(}e_1\mathtt{,}\dots\mathtt{,}e_n\mathtt{)}\,,st) &\triangleq \mathcal{C}(e_1\,,st)\,@\cdots@\,\mathcal{C}(e_1\,,st)\,@\,[\mathtt{ITuple}(n)] \\
\mathcal{C}(\mathtt{fn}\;\mathtt{(}x\mathtt{:}T\mathtt{)} \rightarrow e\,,st) &\triangleq [\mathtt{IClosure}(\mathcal{C}(e\,,x :: st)\,@\,[\mathtt{IPopEnv}])] \\
\mathcal{C}(\mathtt{fun}\,f\,\mathtt{(}x\mathtt{:}T\mathtt{)}\,\mathtt{:}\,T' \mathtt{=} e\,,st) &\triangleq [\mathtt{IRecClosure}(\mathcal{C}(e\,,x :: f :: st)\,@\,[\mathtt{IPopEnv}])] \\
\mathcal{C}(e \mathtt{=} e'\,,st) &\triangleq \mathcal{C}(e\,,st)\,@\,\mathcal{C}(e'\,,st)\,@\,[\mathtt{IEcon;IYes}] \\
\mathcal{C}(e \mathtt{\#} e'\,,st) &\triangleq \mathcal{C}(e\,,st)\,@\,\mathcal{C}(e'\,,st)\,@\,[\mathtt{IFcon;IYes}] \\
\mathcal{C}(e_1 \parallel \cdots \parallel e_n\,,st) &\triangleq [\mathtt{IBranch}(\mathcal{C}(e_1\,,st),\dots,\mathcal{C}(e_n\,,st))] \\
\mathcal{C}(e \mathtt{\&} e'\,,st) &\triangleq \mathcal{C}(e\,,st)\,@\,[\mathtt{IDiscardStkTop}]\,@\,\mathcal{C}(e'\,,st) \\
\mathcal{C}(\mathtt{let}\;x \mathtt{=} e \;\mathtt{in}\; e'\,,st) &\triangleq \mathcal{C}(e\,,st)\,@\,[\mathtt{IStkToEnv}]\,@\,\mathcal{C}(e'\,,x :: st)\,@ \\
&\qquad [\mathtt{IDiscardEnvTop}] \\
\mathcal{C}(e_1\;\cdots\;e_n\,,st) &\triangleq \mathcal{C}(e_1\,,st)\,@\,\mathcal{C}(e_2\,,st)\,@\,[\mathtt{ICall}]\,@\cdots@\,\mathcal{C}(e_n\,,st)\,@ \\
&\qquad [\mathtt{ICall}] \\
\mathcal{C}(\mathtt{distinct(}e_1\mathtt{,}\dots\mathtt{,}e_n\mathtt{)}\,,st) &\triangleq \mathcal{C}(e_1\,,st)\,@\cdots@\,\mathcal{C}(e_1\,,st)\,@\,[\mathtt{IDistinct}(n)\mathtt{;IYes}] \\
\mathcal{C}(\mathtt{case}\,e\;\mathtt{of}\;\vec{K}\,\vec{x}\;\mathtt{->}\;\vec{e}\,,st) &\triangleq \mathcal{C}(e\,,st)\,@ \\
&\qquad [\mathtt{ICase}(\vec{K}\,,(\mathcal{C}(\vec{e}\,,\vec{x} :: st)\,@\,[\mathtt{IDiscardEnvTop}]))]
\end{aligned}
$$

Figure C.2: Compilation function

symbol table every time we move past a binder, this lookup operation produces the correct de Bruijn index for any given occurrence of a value identifier.

We define a compilation function $\mathcal{C}(e\,,st)$ which takes an αML expression e and a symbol table st and produces a list of bytecode instructions. This function is defined in Figure C.2, where we use the notation $I\,@\,I'$ for the concatenation of two lists of instructions. Given a closed expression e, its implementation as a list of bytecode machine instructions is given by $\mathcal{C}(e\,,[])$ where [] represents the initial symbol table (an empty list).

We will illustrate the workings of the bytecode machine by studying certain clauses of the definition of $\mathcal{C}$. Together with the informal semantics of instructions presented above, this gives an idea of the internal workings of the interpreter. For the interested reader, the source code is available from the author's web page.

- **Abstractions, $\mathtt{<}e\mathtt{>}e'$:** This case illustrates how values are constructed by the bytecode machine. The first part of the listing produces the machine value mv resulting from e and the second part produces the machine value mv' from e'. At this point, mv' and mv are the top two elements of the stack. Finally, the `IAbs` instruction converts these into the single abstraction value $\mathtt{MAbs}(mv\,,mv')$, as required.

- **Recursive functions, $\mathtt{fun}\,f\,\mathtt{(}x\mathtt{:}T\mathtt{)}\,\mathtt{:}\,T' \mathtt{=} e$:** The compilation of a recursive function expression produces a single `IRecClosure` instruction, whose argument is the instructions corresponding to the body of the function. The expression e is compiled with the extended symbol table $x :: f :: st$ which represents the fact that f and x are both bound in the body of the function. The value identifier x is pushed on second because the binding for f (i.e. the closure itself) is added when the closure is created, but the binding for x is added later when the closure

is actually applied to an argument. As mentioned in Section C.1, the final instruction of any function body is `IPopEnv`, which discards the environment of the closure.

- **Sequential composition, e & e':** In a sequential composition the instructions produced by compiling e are executed first, followed by `IDiscardStkTop`, which discards the result of e. Finally, the instructions corresponding to e' are executed and their result is the overall result of the sequential composition.

- `let` **bindings,** `let` x = e `in` e'**:** In a `let` binding the instructions corresponding to e are evaluated first and produce a result mv at the top of the stack. The instruction `IStkToEnv` transfers mv to the top of the environment, which is equivalent to substituting that value for x throughout e'. The instructions produced by compiling e' are then executed, and finally `IDiscardEnvTop` removes mv from the top of the environment because we are leaving the scope of the x binder.

- `case` **expressions,** `case` e `of` $\vec{K}\,\vec{x}$ -> $\vec{e}$**:** Figure C.2 uses a shorthand for the syntax of `case` expressions to save space. The notation $\vec{K}\,\vec{x}$ -> $\vec{e}$ is meant to suggest a finite list of single clauses $K_i\,x_i$ -> e_i. Compilation of `case` expressions is relatively straightforward: first the instructions from e are executed, followed by `ICase`. The arguments of the `ICase` instruction are the results of compiling the expressions from the various clauses of the `case` expression, each time with the symbol table extended appropriately. In each case, the final instruction is `IDiscardEnvTop` as we are leaving the scope of a binder.

As noted above, we have not provided a formal semantics for the execution of bytecode machine instructions. Therefore we can only conjecture a safety result for the αML bytecode machine.

Conjecture C.2.1 (Safety for the bytecode machine). *Suppose that e is a well-typed αML expression, i.e. that $\emptyset \vdash e : T$. Then, if $\mathcal{C}(e\,,[\,]) = I$ then the execution of the instruction listing I does not get stuck.* ◇

The correctness of the compilation process and of the bytecode machine rely on the fact that the input instructions come from the compilation of closed, well-typed expressions. For example, the projection instruction `IProj`(i) gets stuck unless the machine value at the top of the stack is `MTuple`$(mv_1, \ldots, mv_n)$ (where $1 \leq i \leq n$) or `MEvar`(k) (for some k). We believe that the αML typechecker rules out any expressions which would compile to such bad lists of instructions, but without an operational semantics we cannot prove this, or even formalise statements like "does not get stuck".

Appendix D

Using the interpreter

In this appendix we present an example of real αML code and discuss the pragmatics of interacting with the αML toplevel. The program is presented in a "literate programming" style, with code fragments interspersed with comments in prose.

D.1 An example program: System F

In this section we will present an αML encoding of the type system of System F, which was presented as an α-inductive definition in Figure 2.4 in Chapter 2. We will extend the example to model the small- and big-step operational semantics of closed System F terms.

```
nametype var and tyvar;;

datatype type = TyVar of tyvar
              | FunTy of type * type
              | ForAll of [tyvar]type
     and tenv = Nil of unit
              | Cons of (var * type) * tenv
     and term = Value of value
              | Var of var
              | App of term * term
              | Spec of term * type
    and value = Lam of type * [var]term
              | Gen of [tyvar]term;;

relation TYPE <: tenv * term * type
and SMALLSTEP <: term * term
  and BIGSTEP <: term * value;;
```

We begin with the datatype declarations for System F terms. These are almost identical to the nominal signature $\mathcal{F}$ presented in Section 2.1. The main difference is that the grammar of terms has been stratified to include a subgrammar of values. As we shall see below, this allows the operational semantics to be encoded elegantly. The lack of built-in list datatypes in αML forces us to define type environments explicitly as a list-like data structure.

The final declaration gives the arities of the inductive relations which we are going to define. The atomic formula `TYPE(gamma,m,t)` corresponds to the typing judgement $\Gamma \vdash M : \tau$. The small-step transition relation $M \longrightarrow M'$ (for closed terms M and M') is represented using the

new relation SMALLSTEP(m,m'). The big-step transition relation $M \longrightarrow^* v$, which reduces a closed term M to a value v (or diverges), is represented as BIGSTEP(m,v).

Astute readers may have noticed that the ttsub relation from Figure 2.4 is missing from this declaration. To demonstrate the functional logic programming capabilities of αML we will refactor the definition slightly so that substitution is defined as a recursive function rather than an inductive relation. This is more natural given functional definitions of substitution in informal mathematics. We now present αML code for substitution in System F.

```
let rec ttsub (t:type) (a:tyvar) (t':type) : type = case t of
  TyVar b -> (a=b & t') || (a#b & TyVar b)
| FunTy tpr -> let t1 = (tpr.1) in let t2 = (tpr.2) in
               FunTy((ttsub t1 a t'),(ttsub t2 a t'))
| ForAll tbnd -> unbind tbnd as <b>t'':[tyvar]type in
                 b#(a,t') & ForAll<b>(ttsub t'' a t');;

let rec mtsub (m:term) (a:tyvar) (t:type) : term = case m of
  Var x -> Var x
| Value v -> (case v of
    Lam z -> let ty = z.1 in unbind z.2 as <x>m':[var]term in
             let ty' = ttsub ty a t in
             let m'' = mtsub m' a t in Value(Lam(ty',<x>m''))
  | Gen z -> unbind z as <b>m':[tyvar]term in b#(a,t) &
             let m'' = mtsub m' a t in Value(Gen(<b>m'')))
| App mpr -> let m1 = mpr.1 in let m2 = mpr.2 in
             App((mtsub m1 a t),(mtsub m2 a t))
| Spec pr -> let m' = pr.1 in let t' = pr.2 in
             Spec((mtsub m' a t),(ttsub t' a t));;

let rec mmsub (m:term) (x:var) (v:value) : term = case m of
  Var y -> (x=y & Value v) || (x#y & Var y)
| Value v -> (case v of
    Lam z -> let ty = z.1 in unbind z.2 as <y>m':[var]term in
             y#(x,v) & Value(Lam(ty,<y>(mmsub m' x v)))
  | Gen z -> unbind z as <b>m':[tyvar]term in
             Value(Gen<b>(mmsub m' x v)))
| App mpr -> let m1 = mpr.1 in let m2 = mpr.2 in
             App((mmsub m1 x v),(mmsub m2 x v))
| Spec pr -> let m' = pr.1 in let t' = pr.2 in
             Spec((mmsub m' x v),t');;
```

The intended meanings of the ttsub, mtsub and mmsub functions are as follows:

- ttsub t a t' implements $\tau[\tau'/a]$, substituting a type for free occurrences of a type variable throughout a type;
- mtsub m a t implements $M[\tau/a]$, the substitution of a type for free occurrences of a type variable throughout a term; and
- mmsub m x v implements the standard notion of substituting a value for free occurrences of a term variable in a term, i.e. $M[v/x]$.

The implementations of these functions are fairly straightforward instances of typed functional programming. Some aspects, however, are worthy of comment:

- The base cases of `ttsub` and `mmsub`, where substitution actually takes place, require a binary branch: the two names in question are either equal or distinct. Unlike in FreshML, this test cannot be implemented using a name-equality test in αML. If the names are not sufficiently constrained it is equally valid to assert either that `a=b` or that `a#b`, and in this case there really are two possible computations to try. One would hope, for efficiency reasons, that this is rarely the case.

- When a substitution is pushed beneath an abstraction, as in the `ForAll` case of `ttsub`, freshness constraints are needed to prevent capture. Unlike in FreshML, the αML `unbind` operation does not generate a globally-fresh name. Therefore we must manually assert the appropriate freshnesses. In this case, the bound type variable `b` should be distinct from the type variable `a` that we are substituting for, and fresh for the type `t'` that is being subtituted in. We express this as the single freshness constraint `b#(a,t')`.

- Term deconstructors in αML (projection, `case` expressions and unbinding) can only remove a single outermost constructor at a time. Therefore there is some noise caused by the need to manually deconstruct terms one constructor at a time (in the `Lam` case of `mtsub`, for example). As discussed in Section 8.1.2, this problem could be alleviated by the inclusion of a more general pattern-matching operation that could be compiled down to a nesting of simpler deconstruction operations.

In keeping with our functional logic style, we define lookup of a variable within a type environment as a partial function.

```
let rec find (gamma:tenv) (x:var) : type = case gamma of
  Nil _ -> no type
| Cons z ->  let y = z.1.1 in let t = z.1.2 in let gamma' = z.2 in
   (x=y & t) || (x#y & find gamma' x);;
```

The expression `no type` is evaluated when the contents of the environment have been exhausted, which causes the current branch of computation to fail immediately. The inductive case uses a branching construct similar to that in the base cases of the substitution functions to test whether we have found the correct variable.

Having defined our helper functions we can now present αML code for the inductively-defined typing and transition relations, which uses the extended syntax for inductive definitions described in Section 7.2.

```
let plc = {{
 (find gamma x) = t
--------------------- [t_var where gamma:tenv, x:var, t:type]
 TYPE(gamma,Var x,t)

  x # gamma  &  TYPE(Cons((x,t),gamma),m,t')
--------------------------------------------- [t_abs where ...]
 TYPE(gamma,(Value(Lam(t,<x>m))),FunTy(t,t'))

 TYPE(gamma,m1,FunTy(t',t))  &  TYPE(gamma,m2,t')
------------------------------------------------- [t_app where ...]
           TYPE(gamma,App(m1,m2),t)

       a # gamma  &  TYPE(gamma,m2,t2)
```

```
------------------------------------------------ [t_gen where ...]
  TYPE(gamma,(Value(Gen <a>m2)),(ForAll <a>t2))

 TYPE(gamma,m,(ForAll <a>t1))  &  (ttsub t1 a t2) = tres
---------------------------------------------------------- [t_spec ...]
            TYPE(gamma,Spec(m,t2),tres)
```

This code is a near-verbatim transcription of the typing rules presented in the α-inductive defintition from Figure 2.4. This is one of the main strengths of αML: it is very easy to transcribe rule-based inductive definitions directly from papers into executable αML code, so we can rapidly produce an executable prototype from a semi-formal mathematical specification.

The biggest difference is the need for type annotations for all variables appearing in every inductive rule. These are needed by the typechecker, but for reasons of space most are omitted in the code fragments presented here. Optionally, the rules can be given names, such as `t_var` here.

```
         SMALLSTEP(m1,m1')
----------------------------------- [ss_app1 where m1,m2,m1',m2':term]
 SMALLSTEP(App(m1,m2),App(m1',m2))

              SMALLSTEP(m2,m2')
------------------------------------------- [ss_app2 where ...]
 SMALLSTEP(App(Value v,m2),App(Value v,m2'))

          SMALLSTEP(m1,m1')
----------------------------------- [ss_spec where m1,m1':term, t:type]
 SMALLSTEP(Spec(m1,t),Spec(m1',t))

             (mmsub m x v) = mres
------------------------------------------------ [ss_beta where ...]
 SMALLSTEP(App(Value(Lam(t,<x>m)),Value v),mres)

          (mtsub m a t) = mres
--------------------------------------- [ss_tybeta where ...]
 SMALLSTEP(Spec(Value(Gen <a>m),t),mres)
```

These small-step transition rules bear a close resemblance to standard presentations of the transition rules for a call-by-value version of System F. The first three rules are the context reduction rules, and the subgrammar of values lets us easily specify that certain rules should only apply when a particular term can no longer be reduced (i.e. when it is a value). The rules with computational content are `ss_beta` and `ss_tybeta`: these are the applications of λ-abstractions and type abstractions to values and types, respectively. They make use of the `mmsub` and `mtsub` functions to perform the appropriate (capture-avoiding) substitution.

```
          yes
----------------------- [bs_val where v:value]
  BIGSTEP(Value v, v)

  SMALLSTEP(m,m')  &  BIGSTEP(m',v)
-------------------------------------- [bs_term where m,m':term,v:value]
```

```
        BIGSTEP(m, v)
}};;
```

The final part of the System F implementation is the definition of big-step transition rules. The `bs_term` rule repeatedly uses the rules of the `SMALLSTEP` relation to reduce the term until it reaches a value, at which point the derivation is completed using rule `bs_val`. Note that the `BIGSTEP` relation may diverge if there is no finite sequence of small-step reductions which lead to a value.

This code illustrates the need for the subgrammar of values. An alternative specification of the big-step reduction relation is that M evaluates to M' when $M \longrightarrow^* M' \not\longrightarrow$, as presented in (Aydemir et al., 2005). The judgement $M' \not\longrightarrow$ would hold when no $\longrightarrow$ transition is possible from M'. The problem is that the αML language presented here does not include a negation operator so we cannot easily encode a predicate such as $\not\longrightarrow$ (see Section 8.1.2). Using a subgrammar of values is one way to detect when no more reductions are possible from a particular term.

D.2 Interacting with the toplevel

Having implemented the System F type system and operational semantics, we now present an example session with the toplevel interpreter to demonstrate its use.

```
> %use "systemf.aml";;
nametype var
     and tyvar
datatype type = TyVar of tyvar
              | FunTy of type * type
              | ForAll of [tyvar]type
and tenv = Nil of unit
         | Cons of (var * type) * tenv
and term = Value of value
         | Var of var
         | App of term * term
         | Spec of term * type
and value = Lam of type * [var]term
          | Gen of [tyvar]term
relation TYPE <: tenv * term * type
     and SMALLSTEP <: term * term
     and BIGSTEP <: term * value
val ttsub : type -> (tyvar -> (type -> type)) = <fun>
val mtsub : term -> (tyvar -> (type -> term)) = <fun>
val mmsub : term -> (var -> (value -> term)) = <fun>
val find : tenv -> (var -> type) = <fun>
val plc : rel -> prop = <fun>
> let a = some tyvar;;
val a : tyvar = a
> let x = some var;;
val x : var = x
> let id = Value(Gen <a>(Lam(TyVar a, <x>Var x)));;
                     ^^^^^^^^^^^^^^^^^^^^^^^^^^^
Type error at line 1, characters 20-47
Found [tyvar]value but expected [tyvar]term
> let id = Value(Gen <a>(Value(Lam(TyVar a, <x>Var x))));;
```

```
val id : term = Value Gen <a>Value Lam (TyVar a, <x>Var x)
> let t = some type;;
val t : type = t
> plc(TYPE(Nil(), id, t));;
- : prop = yes
Type ',' (then Enter) to look for more answers, or '.' to stop.
.
> t;;
- : type = ForAll <?h>FunTy (TyVar ?l, TyVar ?l)
```

The first command loads the definition of the SystemF datatypes, functions and relations from the file `systemf.aml`. The interpreter responds with the datatypes that have been declared and the types of the various functions defined in the file. Then the user creates a type variable `a` and a term variable `x` and uses them to build an expression `id` corresponding to the polymorphic identity function $\Lambda\alpha.\, \lambda x{:}\alpha.\, x$. However, they make a mistake and the expression is rejected by the typechecker. The error message makes it clear what has gone wrong: they have forgetten that `Lam` is a `value`, not a `term`. The corrected version of `id` passes the typechecker.

The user then wishes to find out the type of `id` in the empty typing context. This should exist, since the System F function $\Lambda\alpha.\, \lambda x{:}\alpha.\, x$ is closed. To find out the type, the user creates an existential variable `t` to stand for the unknown type. They then pass the query term `TYPE(Nil(), id, t)` into the `plc` function, which is the encoding of the typing and operational semantic rules as an αML recursive function. The system replies `yes` to indicate that a solution has been found. The user is then presented with a choice: they can either accept the solution (and the corresponding instantiation of `t`) or discard it and keep looking for another. The limitations of a text-based terminal interface are particularly apparent here, particularly as the user cannot investigate a solution before deciding whether to accept it. In our example, the user elects to stop searching, then evaluates the expression `t` to find out how it has been instantiated. This produces the answer "`ForAll <?h>FunTy (TyVar ?l, TyVar ?l)`" which looks slightly suspicious as the type of the polymorphic identity function should be $\forall\alpha.\, \alpha \to \alpha$. The user wishes to investigate further and takes a look at the current constraints recorded in the interpreter, with the following result.

```
> %constraints;;
Substitution constraints:
{t = ForAll <?h>FunTy (TyVar ?l, TyVar ?l),
 ?d = FunTy (TyVar ?l, TyVar ?l),
 ?g = Value Lam (TyVar ?l, <?m>Var ?n),
 ?i = Nil (),
 ?j = TyVar ?l,
 ?k = <?m>Var ?n,
 ?o = TyVar ?l,
 ?p = TyVar ?l,
 ?q = Var ?s,
 ?r = Nil (),
 ?t = ?s,
 ?u = TyVar ?l,
 ?v = ?s,
 ?w = Cons ((?s, TyVar ?l), Nil ())}
Other constraints:
{<(?h:tyvar)>(?l:tyvar) = <(a:tyvar)>(a:tyvar),
 <(?m:var)><(?h:tyvar)>(?n:var) = <(x:var)><(a:tyvar)>(x:var),
```

```
<(?s:var)>(?s:var) = <(?m:var)>(?n:var)}
```

The system replies with a long list of constraints. They are divided into two lists: the first of "defining equations" for certain existential variables and the second of "other constraints" between variables of name sort. The "other constraints" are subject to potentially non-deterministic search during the constraint solving process. While scanning the list, the user's eye is drawn to the third constraint from the bottom:

```
<(?h:tyvar)>(?l:tyvar) = <(a:tyvar)>(a:tyvar)
```

which implies that `?h` and `?l` must always represent the *same* type variable—an example of aliasing. Therefore, we can deduce that the inferred type of `id` is really

```
ForAll <?h>FunTy (TyVar ?h, TyVar ?h)
```

as expected. This example highlights a deficiency of the toplevel interface: when answers are pretty-printed to the user, not all of the information in the constraints is used. The "substitution constraints" mentioned above are used to replace existential variables by their instantiations but other constraints between names are not used. This could be addressed in a future version of the software and would produce a more usable user interface.