Technical Report

UCAM-CL-TR-618
ISSN 1476-2986

Number 618

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

The Fresh Approach: functional programming with names and binders

Mark R. Shinwell

February 2005

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

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

This technical report is based on a dissertation submitted December 2004 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Queens' 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

Abstract

This report concerns the development of a language called Fresh Objective Caml, which is an extension of the Objective Caml language providing facilities for the manipulation of data structures representing syntax involving α-convertible names and binding operations.

After an introductory chapter which includes a survey of related work, we describe the Fresh Objective Caml language in detail. Next, we proceed to formalise a small core language which captures the essence of Fresh Objective Caml; we call this *Mini-FreshML*. We provide two varieties of operational semantics for this language and prove them equivalent. Then in order to prove correctness properties of representations of syntax in the language we introduce a new variety of domain theory called *FM-domain theory*, based on the permutation model of name binding from Pitts and Gabbay. We show how classical domain-theoretic constructions—including those for the solution of recursive domain equations—fall naturally into this setting, where they are augmented by new constructions to handle name-binding.

After developing the necessary domain theory, we demonstrate how it may be exploited to give a monadic denotational semantics to Mini-FreshML. This semantics in itself is quite novel and demonstrates how a simple monad of continuations is sufficient to model dynamic allocation of names. We prove that our denotational semantics is computationally adequate with respect to the operational semantics—in other words, equality of denotation implies observational equivalence. After this, we show how the denotational semantics may be used to prove our desired correctness properties.

In the penultimate chapter, we examine the implementation of Fresh Objective Caml, describing detailed issues in the compiler and runtime systems. Then in the final chapter we close the report with a discussion of future avenues of research and an assessment of the work completed so far.

Contents

1 Introduction

'Imagination is more important than knowledge.' —Einstein

THIS REPORT is about two things: *names* and *metaprogramming*.

Names are ubiquitous in the field of computer science and throughout today's networked world. Important elements of computer systems such as files, disks, machines and networks are all identified by names. Programs and machines often communicate over virtual 'channels' which are identified by names[34]. When writing programs, use is made of variables, file handles, process identifiers and many other sorts of name in disguise.

All of these names may be divided[39, 35] into two varieties: *pure* names and *impure* names. A name which is said to be pure is just a handle on something. Given a pure name, we can inspect what it is referring to and check to see if it is the same name as another. However, usually that is all and no further operations are permitted. In contrast, an impure name has some structure *actually within the name* which we can examine. An example might be the electronic mail address `mrs30@cam.ac.uk` or the Internet address of a computer, say `131.111.8.42`. In each of these cases, there is some user-visible structure to the name—in these examples, the structure is defined by international standards. The names justify their designation because they still refer to something: a mailbox and a computer respectively.

This report deals with the use of pure names in *metaprogramming systems*. Metaprogramming is the art of constructing software which manipulates the syntactic structures of other programs. There are many large-scale examples in common use today: compilers, interpreters and theorem provers being good examples. Metaprogramming can be *homogenous*, meaning that the metaprogram is written in the same language as the program being manipulated; or *heterogeneous*, meaning that these languages may differ. Sheard[58] provides a good survey of these and other issues in metaprogramming.

We shall be particularly interested in possibly-heterogeneous metaprogramming in *functional* languages such as Haskell, Standard ML, Objective Caml and so forth. These languages are particularly good for metaprogramming due to a few salient features, of which the following two are arguably the most important.

► *User-defined datatypes and pattern-matching.* It is straightforward to represent the syntax of some target language (known as the *object language*) by defining a *datatype* to represent the various shapes of syntactic structure which occur in that language. Values of such datatypes are created by parsing a program's *concrete syntax* into *abstract syntax*[30]. A portion of abstract syntax will likely be tagged to say what it is (a variable, a conditional statement, etc.) and it may well contain smaller syntactic entities within it. Functional programming languages make it easy to construct values representing abstract syntax and then pick them apart again using pattern-matching. These features tend to save a large amount of tedious coding and help with the readability of metaprograms.

► *Definition by structural recursion.* Functional languages permit functions over syntax to be defined by *structural recursion*[8]. Typically a program will make use of such recursion to manipulate abstract syntax trees by traversing through the various structures and performing different actions depending on the type of entity encountered along the way. Many such transformations can be captured using the idea of a *catamorphism*[31], a generalisation of the traditional `foldl` and `foldr` functions over lists.

The writing of program-manipulating programs entails the writing of name-manipulating programs. Otherwise, we would have to restrict our attention to singularly uninteresting languages involving no notion of 'name' whatsoever. A prime concern in metalanguage design should therefore be the provision of suitable facilities for manipulating names. On the surface this might not seem a difficult problem: after all, names at first sight do not appear to be

particularly complicated things. In fact, they are not—it is the *operations* in which they are involved which give rise to the complexities.

The specific complications which we shall deal with in this report are those arising from the *binding* of names by language constructs. Here are three short examples—one from mathematics, one from Objective Caml and one from the λ-calculus[5]—which illustrate binding operations.

$\int_0^1 f(x,y)\ dx$	an integral
`let x = 42 in x * x`	local declaration
$\lambda x.\ \lambda y.\ x\ y$	function abstraction

Name binding is as ubiquitous as names are themselves and poses significant problems for the representation of object language syntax. The nub of the problem is that in pencil-and-paper practice—when designing syntax-manipulating algorithms in the object language or maybe proving theorems about them—we often identify bound names *up to* α*-conversion*. For example, $\lambda x.\ \lambda y.\ x\ y$ is α-convertible to both $\lambda x.\ \lambda z.\ x\ z$ and $\lambda y.\ \lambda x.\ y\ x$. This process of renaming variables to produce α-variants of terms is often essential during the manipulation of syntax where terms may be substituted into each other, or in other scenarios where there is a possibility of name clashes. A classic example is the algorithm of *capture-avoiding substitution* which is used in the definition of β-reduction for the λ-calculus. Free-and-easy application of α-conversion during pencil-and-paper work captures the idea that the *particular* bound names used should not somehow matter, so long as they are kept suitably distinct from each other. This convention that free and bound names should always be kept disjoint and distinct from each other is often known as the 'Barendregt variable convention'[5].

The mechanisation of algorithms which involve the manipulation of syntax with binders is made difficult and time-consuming by the necessity to incorporate code to handle these scenarios. In this report, we are concerned with extending a functional programming language (Objective Caml) with core features which assist these metaprogramming tasks. The resulting language, Fresh O'Caml, provides for the generation of fresh names which we can use to represent object language names which may be involved in binding operations: such names will be referred to as *bindable* names. It also assists us with the representation of object language binding constructs by providing facilities to package up syntax trees using *abstraction expressions*. When deconstructing the values of such expressions, the runtime system ensures that names inside are freshened up accordingly to avoid clashes. Additionally, some extra utility constructs are provided to increase the expressiveness of the language.

Fresh O'Caml is the successor to the FreshML language[63, 49, 18]—a design which yielded an interpretive system accepting a syntax based on the Core language of Standard ML with added features for manipulating names. Early versions of the language, now referred to as FreshML-2000, used a complicated static type system[49] which was thought to be necessary to enforce certain correctness properties. (We give a detailed discussion of this type system in §6.8.) Later versions used a much-simplified type system, which gave rise to more expressive versions of the language and finally led onto the development of Fresh O'Caml and the subsequent deprecation of the FreshML system. The FreshML interpretive system is now unsupported.

What properties are desirable of a name-manipulating solution for a metaprogramming language? We believe the following aspects are important. For each, we give a brief description of how Fresh O'Caml addresses the issue.

▶ *Simplicity.* The solution should provide the essential functionality—the generation of fresh names together with the construction and deconstruction of representations of object language syntax with binding—in the most straightforward way possible. This assists the programmer who has to use the facilities and the theorist who wishes to prove properties about them. In Fresh Objective Caml we provide fairly general, *first order* support for representing binding operations. The new facilities are straightforward to understand and to use.

▶ *Correctness.* The additional facilities should have a solid mathematical foundation, together with the necessary proofs, to make it convincing to a user that the representations of syntax which they are using are in some sense correct. In this report, we present a full proof of some correctness properties of a core language Mini-FreshML which captures the essence of Fresh

O'Caml—showing how contextual equivalence classes[1] of values and expressions correspond in a certain way to α-equivalence classes of object language syntax.

► *Readability.* Programs written using the new solution should bear a good correspondence to the underlying algorithms. Firstly, there are the usual software engineering arguments which we shall not delve into here. Secondly, the sort of algorithms often encoded as metaprograms are typically developed on paper, often with some good theoretical foundation. It is a desirable property of a metalanguage to be able to encode such algorithmics so that the resulting metaprogram bears a good correspondence to the original statement of the algorithm. As will be seen in the next chapter, Fresh O'Caml programs encoding syntax-manipulating algorithms often bear a close resemblance to descriptions of the algorithms themselves.

► *Integration.* New language constructs introduced by the solution should integrate well with the existing language features, for example unbounded fixpoint recursion. It is also desirable that the new language constructs should fit into the existing typing system of the programming language in order to ease implementation.

► *Efficiency.* The solution should be designed in such a way as to permit efficient implementation. At the current time, the Fresh O'Caml implementation is at a sufficiently early stage to still be lacking in this area: however, later on in this report we discuss the next avenues to be taken in order to improve this.

This report provides detailed description and discussion of Fresh O'Caml, along the way encompassing a whole spectrum of computer science. For the pragmatist there is detailed discussion of the language itself—ranging from a user's perspective through to nitty-gritty details of the runtime system implemented in C. For the theorist, we present a variety of domain theory which is good for reasoning about names and name binding. This forms the basis for a denotational semantics that is sufficiently powerful to establish some strong correctness results about our language—and is also interesting in its own right. But we must emphasise that we are strictly concentrating on functional programming: whilst our work may well be of use in other areas such as logic programming, we do not wish to embark on a moral crusade suggesting that our approach to name binding is universally applicable.

1.1 Other approaches to handle binding

Approaches to the problem of name binding in metaprogramming languages may be partitioned into two: *first-order* and *higher-order*. These designations correspond to the varieties of metalanguage constructs which are used to represent object language binding constructs. The Fresh approach described in this report is first-order: object language binding constructs are not represented by higher-order functions. Other first-order schemes include both simple ones using textual representations of names and those based on *de Bruijn indices*. In contrast, we have the various approaches based on *higher-order abstract syntax* which all exploit the function abstraction constructs of the metalanguage in order to represent object-level binding. Let us examine these differing approaches in more detail.

1.1.1 de Bruijn indices

An oft-cited means of tackling the problems of name binding is the use of *de Bruijn indices*[11, 12, 17] in abstract syntax trees. This is a 'nameless' way of representing object-level syntax via the use of numbered binders: rather than a language extended wholesale with new constructs, it is simply a scheme of representation which can be used with a standard metaprogramming language. It concentrates on the *shape* of the binding structure rather than on particular names. Instead of placing a bound name in a syntax tree, we place an integer which identifies how many levels of binders we should skip back over in order to find the one which binds at the point of interest. For example, we could represent the term $\lambda x.\ \lambda y.\ y\ x$ by the nameless term $\lambda.\ \lambda.\ 0\ 1$. Note how names on the binders themselves become redundant.

Whilst de Bruijn indices do indeed capture the idea that 'actual names of bound variables should not matter', they are inconvenient in practice. Code which manipulates nameless representations is likely to be difficult to understand and maintain. One particular problem is

[1] Two expressions are said to be contextually equivalent if they may be freely exchanged within the source text of some enclosing program without affecting the program's observable results.

that replacing one subterm with another inside some larger term may in general result in the de Bruijn indices having to be adjusted. Consider for example the following code fragment, where the ellipsis stands for some large function body.

```
let f = fun x -> ... in let g = fun y -> y + (f y) in fun z -> (f, g)
```

We might wish to perform a transformation on this code fragment to inline the function `g`, yielding the following.

```
let f = fun x -> ... in fun z -> (f, fun y -> y + (f y))
```

Let us consider using de Bruijn indices to represent these pieces of syntax. Assuming that '`let`' is a non-recursive binding construct, then a representation of the first fragment above could look like the following.

```
let _ = fun _ -> ... in let _ = fun _ -> 0 + (1 0) in fun _ -> (2, 1)
```

Applying the inlining transformation, we then obtain the following.

```
let _ = fun _ -> ... in fun _ -> (1, fun _ -> 0 + (2 0))
```

Note that two extra things have had to happen here: firstly, the removal of the binding for `g` means that the first component of the pair needs a de Bruijn index adjusting. A similar thing happens in the second component, where the inlining of the body of `g` has caused it to be in a different place with respect to the various binders.

This simple example shows how a 'nameless' style of programming can be difficult to work with. Fresh O'Caml provides a 'nameful' style, where the programmer can gain access to concrete names whilst still having the reassurances provided by capture-avoidance and freshness properties.

1.1.2 Threading of name states

Peyton-Jones and Marlow[41] present a good overview of the problems surrounding name binding in the Glasgow Haskell Compiler inliner. Early versions of that compiler blindly renamed every bound variable to a fresh one in order to avoid problems of name capture. However, not only is this inefficient due to the gratuitous 'churning' of names, as those authors put it, but also cumbersome due to the lack of side-effecting constructs in Haskell: some kind of name state must be threaded through all algorithms dealing with the generation of fresh names. The authors describe how more modern versions of the compiler cope with this by adapting algorithms such as those for capture-avoiding substitution so they take note of all names currently in scope. This enables the generation of fresh names to be minimised and there is no gratuitous renaming. When a fresh name is required, a probabilistic approach is used to (hopefully rapidly) find a suitable one.

1.1.3 Higher-order abstract syntax

Unlike Fresh O'Caml, *higher-order abstract syntax* (HOAS)[43, 24] exploits the function abstraction constructs of the metalanguage to represent object-level binding, yielding another 'nameless' style of representation. As an example, a HOAS representation of abstract syntax trees of the λ-calculus could be written as

```
type lam = Lam of lam -> lam | App of lam * lam;;
```

Whilst such an embedding at first seems to be rather elegant, it does not fit well in the framework of a general-purpose functional programming language. The good points about it are that capture-avoidance—and indeed object-level substitution—come for free, by virtue of the properties of meta-level function application. (Having said that, an argument that 'object-level substitution for free' is highly desirable does not really hold its ground for a language such as Fresh O'Caml, where such operations are easily defined. See for example our code for capture-avoiding substitution in §2.2.3: it is straightforward and fast to write.)

The serious concern about such declarations, however, is the contravariant occurrence of the type `lam`. This means that the datatype is a mixed-variance inductive definition, precluding the construction of functions over it by the usual recursive techniques. For example,

when specifying a transformation over such a datatype using a catamorphism (say to pretty-print terms using a function of type `lam -> string`) then one must also provide the inverse *anamorphism* (in this case a function of type `string -> lam`, which amounts to writing a parser).

Moreover there are sufficiently many elements of the function space `lam -> lam` that many of them, when encapsulated inside `lam`'s constructors, do not correspond to legal abstract syntax trees of the object language. The ones that do not are the so-called 'exotic terms', for example:

```
let exotic =
  Lam (fun t -> match t with Lam _ -> t | App _ -> App (t, t));;
```

Whilst this is a typeable expression of type `lam`, it does not correspond to the abstract syntax of any λ-term.

Another unfortunate property of HOAS representations is that since a value representing a term may contain *computations* at the binding points, side effects will be delayed until the term is deconstructed via function applications. This may not be in keeping with a strict metalanguage such as Objective Caml.

In an attempt to deal with the problem of exotic terms there are type systems[15] to restrict the values which may be passed as the arguments to constructors such as `Lam`. These work by using the observation that so long as the types of such constructors only admit suitably *parametric* functions[69], then no exotic terms arise. However, such restrictions produce a technically-complex framework which still has inadequacies—as seen in the conclusion of [15] where the authors note that an algorithm implementing a simple structural equality test is not definable in their system.

Another alternative[14] is to adopt a 'weak' rather than the 'strong' HOAS embedding which we saw above. This yields type declarations such as the following:

```
type lam = Lam of var -> lam | App of lam * lam;;
```

for some type `var` of object language identifiers. This produces normal, positive inductive definitions at the expense of doing away with object-level substitution for free. We do, however, still retain the capture avoidance properties which we desire. All begins to look rosy, but there is a further difficulty: how does one define the type `var`? This needs to be suitably abstract, in some sense, so that exotic terms (involving functions of type `var -> lam`) cannot be created by case distinction on values of type `var`. Suitable types for `var` can be defined, but this means that one cannot define the algorithm for capture-avoiding substitution, for example.

It still seems to be the case that there exists no single, integrated solution based on higher-order abstract syntax that provides a full-scale name-manipulating metalanguage along the lines of Fresh O'Caml. Previous work by Miller[33] on an ML-style language $\mathbf{ML}_\lambda$ equipped with function types τ `=>` τ' goes some way to remedy the situation, but again becomes somewhat clumsy when it comes to defining functions by structural induction over abstract syntax. Bindlib[53] provides a framework for manipulating abstract syntax with binders in O'Caml, but it is not fully automatic since user-defined functions must be written for term construction. Another take on the subject is the more recent work of Washburn and Weirich[70], who (in addition to a nice survey of some of these problems) have shown how one can enforce parametricity restrictions equivalent to those of [15] by just using first-class polymorphism in Haskell. However, as those authors identify, their approach still fails to provide any direct access to concrete names.

So whilst HOAS has had nice applications, for example in the areas of logic programming and theorem proving, the first-order approach presented in this report does appear to be more suited for our target application area.

1.2 Other related work

In this section we provide a brief summary of some other work which is related in one way or another to that of our own. Much of this revolves around the underlying permutation model of name binding in *FM-set theory*[20, 18] upon which the FM-domain theory of Chapter 4 is based. This is not an exhaustive list: other related work is cited later in the report where it fits into the main flow of the text.

► *Trees with hidden labels.* The work by Cardelli et al.[9] on manipulating trees with hidden labels must be singled out as having directly affected the course of the FreshML (and thereafter Fresh O'Caml) language design. For it was this work which led to the realisation that it was not necessary to worry about certain side-effects of name generation which arise during the deconstruction of values representing object language binding constructs. This led to the deprecation of a complicated system of static analysis (discussed at length in §6.8).

► *Nominal unification.* As we discussed in §1.1.3, there is a distinct split between Fresh O'Caml and HOAS-based approaches by virtue of Fresh O'Caml representing binding using first-order constructs. This same division between first-order and higher-order approaches is apparent in the work[67] by Urban et al. on *nominal unification*. In that setting, the problem is to take two object-level terms involving binding and determine whether there exists some substitution of terms for variables which makes the two original terms α-equivalent. Those authors' work is again based on the permutation model of name binding, involving access to concrete names rather than using higher-order representations, and therefore draws on much of the same theory as ourselves. The work shows how to produce a decidable, most-general unification algorithm which operates on object-level terms involving binding—and this is done without much of the awkwardness associated with traditional higher-order unification problems.

► *Logic programming.* The work on nominal unification has led to a series of papers on α-Prolog[10], an evolving variant of Prolog which draws on the permutation model of name binding in order to manipulate abstract syntax. It will be interesting to see in the future if that model can be made to fit a logic programming setting as well as it fits a functional one.

► *Theorem-proving.* The implementation of theorem-proving technologies is an important application of metaprogramming. The doctoral thesis of Gabbay[18], which presents the full details of FM-set theory together with a discussion of an early version of FreshML, includes a formalisation of FM-set theory inside the Isabelle proof checker. Further discussion is available elsewhere[19].

► *Game semantics using FM-sets.* Abramsky et al. have produced recent work[2] which describes a fully-abstract model of the nu-calculus[65, 50, 66], based in a setting of game semantics. They exploit one of the key properties of FM-set theory just as we do in this report: it enables all dependencies on parameterising name sets to be made implicit. The work is of particular interest since the nu-calculus is closely related to Mini-FreshML (described in Chapter 3), albeit lacking in recursion.

► *Term rewriting systems.* Work by Gabbay et al.[16] has shown how both first-order and higher-order term rewriting systems may be reformulated into a first-order approach based on FM-set theory. As for Fresh O'Caml and the work on nominal unification, a 'nameful' approach is adopted where bound entities are explicitly named; the resulting rewriting systems respect α-conversion of terms in the correct manner.

► *Name management calculi.* Recent work by Moggi and Ancona[4] presents a monadic calculus for name management using ideas from FM-set theory.

1.3 Report structure

This report consists of seven chapters, of which this is the first. In the next chapter, we describe the extensions to Objective Caml which constitute the Fresh O'Caml language. Chapter 3 then proceeds to formalise a small language which provides the core features of Fresh O'Caml. Two styles of operational semantics, which we prove equivalent, are provided to explain the evaluation strategy of the language. In Chapter 4, we lay the groundwork for a theoretical treatment by providing a variant of classical domain theory specially designed for modelling names. Chapter 5 presents a denotational semantics using this domain theory. It then proceeds to derive observational equivalence results to show in what sense the representations of object language terms are correct. In Chapter 6, we discuss implementation details. Finally in Chapter 7 we survey future work and draw conclusions.

2 Fresh Objective Caml

'A programming language is a tool
that has profound influence on
our thinking habits.' —Dijkstra

WE NOW BEGIN to describe the Fresh Objective Caml language in detail, using examples along the way to illustrate the new language constructs. From this point onwards we assume that the reader is familiar with the standard O'Caml language, from whose distribution the Fresh Objective Caml system[1] is derived. In order to comply with the licence terms of O'Caml, the modifications are distributed as a patch bundled together with a modified `configure` script which silently applies the patch when the distribution is built for the first time. The modifications to the system which form the patch are distributed under a BSD-style licence.

2.1 Representation of object language names

The first aspect which we examine is the typing and dynamic creation of values which represent object language bindable names.

2.1.1 Bindable names: typing

So what types should be assigned to identifiers representing object language bindable names? An initial attempt might be to have a single metalanguage type, perhaps called `name`, for this purpose. However, this is not particularly satisfactory since it would be nice to be able to distinguish between different *varieties* of names which may occur in object language syntax. For example, when representing types and terms of the polymorphic lambda calculus[22, 54, 23, 55] (PLC) we will need to represent type variables and term variables—both of which may take part in binding operations at the object level. If we can distinguish between these varieties of names at the meta level (that is, in the source text of some Fresh O'Caml program), then the compiler will be able to enforce stronger static checks on our source text during type-checking.

Given multiple varieties of names it is desirable to have polymorphism over these varieties. For example, we shall see later that there exists a Fresh O'Caml construct `swap` a `and` a' `in` e, where a and a' must be of the same type of names and e is some expression. If we wish to construct a function such as

```
let map_swap a b = map (fun x -> swap a and b in x);;
```

then we should be able to assign polymorphic types to the function parameters in such a way as to constrain them to be of a type of names—and furthermore, of the *same* type of names.

The best solution to this problem would be to declare some variety of type class whose members are those types which represent bindable names. However, since neither Standard ML nor O'Caml provide such Haskell-style type classes[42] we were forced to adopt an alternative strategy.

The first attempt at a solution was implemented in the FreshML interpretive system based on Standard ML. This provided for multiple types of bindable names using the `bindable_type` declaration. For example,

```
bindable_type var;
```

introduced a new type constructor of zero arity called `var` which could be used to represent a certain variety of bindable names. Multiple such declarations could be issued within the

[1] Available from `http://www.fresh-ocaml.org/`, together with example programs by the author and others.

source text. Then, a subclass of the equality type variables[2] was created known as the *bindable type variables*. These were named `'@a`, `'@b` etc. and provided polymorphism in the desired manner. For example, the FreshML equivalent of the above function `map_swap` would be typed as follows.

```
map_swap : '@a -> '@a -> 'b list -> 'b list
```

During the transition to the newer Fresh O'Caml system a rethink was required, for Objective Caml does not provide the notion of classes of type variables. Instead, we use a single polymorphic type `'a name` and subsume polymorphism over the different kinds of name under the usual mechanisms for polymorphic types. In order to produce types corresponding to specific varieties of name, for example names of variables and names of type variables in PLC, we use generative `type` declarations as follows.

```
type t and var = t name;;
type t and tyvar = t name;;
```

In Fresh O'Caml, `map_swap` now receives the following type.

```
map_swap : 'a name -> 'a name -> 'b list -> 'b list
```

2.1.2 Bindable names: creation

Initially, there exist no values of any bindable type. To create them, one simply uses the `fresh` expression:

```
# let x = fresh;;
val x : 'a name = name_0
```

This declares a new identifier `x` and assigns a fresh *atom* to it. We shall learn rather more about atoms later in this thesis, but for the moment let us just observe that they are the semantic values corresponding to bindable names. No matter what type of bindable names is involved, the atoms are drawn from the same global set at runtime[3].

Values of bindable type are printed as `name_0`, `name_1` etc.; this numbering is on a 'per-response' basis in the same manner as the usual method for renaming type variables to start from `'a` on each response. The actual atoms which are assigned to identifiers of bindable type are completely concealed from the programmer (rather like the way in which the actual *addresses* of reference cells are concealed in O'Caml).

Having made the above declaration, we could use the identifier `x` to represent an object language bindable name. If no explicit type annotations are provided by the programmer then the name is assigned a polymorphic type, in this case `'a name`.

In the following fragment of an interactive session we specify an explicit type annotation for the new name.

```
# let (x:var) = fresh;;
val x : var = name_0
```

From the programmer's point of view, the only thing which may be done with an atom after creation is to test whether it is the same as another. In particular, *there is no ordering on atoms* to be consistent with the underlying FM-set theory[20]. Problems which arise as a consequence of this restriction are discussed in §7.1.2.

2.2 Representation of object language binding

In order to represent object language binding constructs Fresh O'Caml uses *abstraction values*, written $<<v>>v'$ for values v, v'. Such values are assigned *abstraction types* $<<\tau>>\tau'$ (for types τ, τ') and may be constructed using the abstraction-former $<<e>>e'$ (for expressions e, e'). The type τ will usually be a type of bindable names, but it may in fact be any type whose values are comparable (such as an algebraic datatype).

[2] Another example of the poor-man's type class.

[3] Writing $\mathbb{A}$ for the countably infinite set of atoms and $\mathbb{A}_\tau$ for the set of values of type τ `name`, what we are effectively doing is setting up an isomorphism between $\mathbb{A}$ and the disjoint union of the $\mathbb{A}_\tau$, as τ ranges over the Fresh O'Caml type expressions.

```
# let alpha, x, y = fresh, fresh, fresh;;
val alpha : 'a name = name_0
val x : 'a name = name_0
val y : 'a name = name_0
# let t1 = Tgen(<<alpha>>(Tlam(TYvar alpha,
                <<x>>(Tlam(TYvar alpha, <<y>>(Tvar x))))));;
val t1 : term = Tgen <<name_0>>(Tlam (TYvar name_0,
                <<name_1>>(Tlam (TYvar name_0,
                <<name_2>>(Tvar name_1)))))
# let t2 = Tgen(<<alpha>>(Tlam(TYvar alpha,
                <<x>>(Tlam(TYvar alpha, <<x>>(Tvar x))))));;
val t2 : term = Tgen <<name_0>>(Tlam (TYvar name_0,
                <<name_2>>(Tlam (TYvar name_0,
                <<name_1>>(Tvar name_1)))))
# let t3 = Tgen(<<alpha>>(Tlam(TYvar alpha,
                <<x>>(Tlam(TYvar alpha, <<y>>(Tvar y))))));;
val t3 : term = Tgen <<name_0>>(Tlam (TYvar name_0,
                <<name_2>>(Tlam (TYvar name_0,
                <<name_1>>(Tvar name_1)))))
```

Figure 2.1: Encoding some PLC terms in Fresh O'Caml.

2.2.1 Abstraction values: construction

When we write an abstraction expression <<e>>e' in a piece of Fresh O'Caml code, we are telling the compiler that we wish to represent a piece of object language binding. We say that the expression between the double angle brackets is in *binding position* and use the same convention for abstraction values and abstraction types. At runtime, evaluation of e' yields the value representing the body of the abstraction whereas evaluation of e yields that which corresponds to the bound identifier(s). (Often, e will just evaluate to a single atom, but expressions of more complicated types are permitted in binding position as we noted above.) The evaluation of e must yield a value that is comparable (in the Standard ML setting this corresponds to e having an equality type). Failure to adhere to this condition raises a runtime `Invalid_argument` exception. We say a little more about this in §2.2.5.

In a programmer's Fresh O'Caml source code, abstraction types are most likely to arise within `type` declarations corresponding to object language syntax trees. Here we shall continue our running example of the polymorphic lambda calculus (PLC) by providing a suitable declaration to represent syntax trees corresponding to types and terms of that calculus:

```
type typ  = TYvar of tyvar
          | TYfn of typ * typ
          | TYforall of <<tyvar>>typ;;

type term = Tvar of var
          | Tlam of typ * (<<var>>term)
          | Tgen of <<tyvar>>term
          | Tapp of term * term
          | Tspec of term * typ;;
```

We can see that abstraction types have been used in those places where we wish to represent some object-level binding. It is now a straightforward matter to create some syntax trees. For example, consider the PLC terms

$$t_1 \stackrel{\text{def}}{=} \Lambda\alpha.\ \lambda x\ :\ \alpha.\ \lambda y\ :\ \alpha.\ x$$
$$t_2 \stackrel{\text{def}}{=} \Lambda\alpha.\ \lambda x\ :\ \alpha.\ \lambda x\ :\ \alpha.\ x$$
$$t_3 \stackrel{\text{def}}{=} \Lambda\alpha.\ \lambda x\ :\ \alpha.\ \lambda y\ :\ \alpha.\ y.$$

We can encode the terms in a Fresh O'Caml interactive session as shown in Figure 2.2.1. Note

how straightforward it is to encode the binding in the object language terms: abstraction type-formers are tightly integrated with the other type constructors used in algebraic datatypes and the corresponding values are structurally similar to the original PLC terms. It is in fact the case that a Fresh O'Caml datatype is still said to be *algebraic* just when it does not involve any occurrences of the function space constructor `->`: it therefore does include those datatypes involving abstraction types in their construction.

Also, observe how the system numbers the bindable names on a 'per-response' basis as mentioned in §2.1.2. We can see from this example that the numbering is particularly useful to highlight the object-level binding inside each term.

In case the reader is wondering, we take the time now to identify that the abstraction-forming expression `<<–>>–` is *not a binding construct*—unlike in an approach based on higher-order abstract syntax where function abstraction is used to represent object-level binding. Similarly, abstraction values do not bind atoms. We can illustrate that `<<–>>–` is not a binder by considering the Fresh O'Caml expressions

$$\texttt{<<}x\texttt{>>}z \qquad \text{and} \qquad \texttt{<<}y\texttt{>>}z$$

where x, y and z are distinct value identifiers of bindable type. The two expressions would be contextually equivalent if the abstraction-former were a binding construct. However, we can see that it is not by distinguishing the two expressions using the following context:

$$\texttt{let } x \texttt{ = fresh in let } y \texttt{ = fresh in let } z \texttt{ = } x \texttt{ in } [\texttt{–}].$$

At runtime x will be assigned some fresh atom, a_1 say. The same happens for y: call the corresponding atom a_2. Then, z will be mapped to a_1 in the current environment and thus the two semantic values to which $\texttt{<<}x\texttt{>>}z$ and $\texttt{<<}y\texttt{>>}z$ evaluate are $\texttt{<<}a_1\texttt{>>}a_1$ and $\texttt{<<}a_2\texttt{>>}a_1$ respectively, which are clearly not contextually equivalent.

The important correctness property of Fresh O'Caml expressions[4] and values of datatypes such as `term` is that

> *their contextual equivalence classes are in bijection with the α-equivalence classes of the object language terms.*

After some work, we shall formally state and prove such a result for a small language *Mini-FreshML* later in this thesis. Later in this chapter we shall see how this result enables the testing of object level α-equivalence to be subsumed under the usual mechanisms for structural equality testing in Fresh O'Caml.

How much support for specific *patterns* of binding should the language provide? The current version of Fresh O'Caml described here takes a rather 'hands-off' approach, in the sense that it provides a single construct (the abstraction expression) which can be used in many different situations. But suppose we were dealing with an object language providing `let`-expressions which may bind multiple identifiers at once, as illustrated in the following code skeleton. The `let`-bindings may be mutually recursive if marked with `rec`; if not, the bindings are non-recursive.

```
let rec x1 = ...
        x2 = ...
in
  foo (x1, x2);;
```

Let us think about the binding forms we have here. If the `let` is to be non-recursive, then the relevant fragment of a Fresh O'Caml `type` declaration called `term` to represent such syntax would be something like:

```
Elet of term list * (<<var list>>term)
```

for some type of bindable names `var` corresponding to object language identifiers. Note how we use a more complicated abstraction type in order to signify that we want to represent the simultaneous binding of multiple identifiers. The `term list` part corresponds to the right-hand side of the various `let`-bindings and the scope of the object-level binding is therefore

[4] Not involving side-effects from references, exceptions and the like.

reflected in the type of the constructor argument. More specifically, the various identifiers introduced in the `let` are not bound within the first component of the pair.

If the binding is recursive, however, this scheme is somewhat unsatisfactory. For in this case, the scope of the identifiers must include the `let`-bindings themselves. In this case, the constructor declaration should look like this:

```
Elet of <<var list>>(term list * term)
```

Now, whilst this could be used to produce a correct (in some sense) representation of the above object language syntax, the question arises as to whether Fresh O'Caml should provide a more specific binding form which could be useful here. For there is nothing currently to stop the programmer erroneously constructing a value where the lengths of the `var list` and `term list` parts are different. One solution, of course, would be to use a dependently-typed language: alternatively, we could introduce further binding forms into Fresh O'Caml to cope with such a situation.

Our view is that whilst more specific forms would be desirable in certain situations in order to increase the potential for stronger static checks, the aim of Fresh O'Caml should be to provide a general framework for handling name binding. As we noted in Chapter 1, there is something pleasing about keeping new constructs simple and straightforward from both a user's and a compiler writer's point of view. Once one starts down the road of incorporating somewhat 'subsidiary' features into the language, there is a danger that it will become over-complicated. For example, if we were to incorporate specific constructs to handle the above situations then how about one which allows mixed recursive/non-recursive bindings too? Finally, it should not be forgotten that our extensions to O'Caml are based on a solid mathematical foundation: multitudinous language constructs do not help to keep the theory and the implementation in step and often just work to obscure the real crux of the matter in the mathematics.

2.2.2 Abstraction values: typing

Given expressions e and e' of types τ and τ' respectively then the abstraction expression $\texttt{<<}e\texttt{>>}e'$ simply has type $\texttt{<<}\tau\texttt{>>}\tau'$. In other words, the type inference algorithm functions in exactly the same way as for pair types.

2.2.3 Abstraction values: deconstruction

So what can we actually *do* with an abstraction value after creation? Such a value can be thought of as a 'black box' whose innards may only be examined via one specific means of deconstruction: matching against an *abstraction pattern*. The lack of any other means of inspecting abstraction values ensures that they may only be deconstructed in such a way as to maintain the Barendregt variable convention. As a result, the programmer can write syntax-manipulating code without having to worry about possible name clashes[5].

Let us see some concrete examples. Here is a function, defined by structural recursion in the usual manner, which performs *capture-avoiding* substitution of the term `t` for the variable `x` throughout the PLC term `t'`.

```
let rec subst t x t' =
  match t' with
    Tvar y -> if x = y then t else t'
  | Tlam (ty, <<y>>t'') -> Tlam (ty, <<y>>(subst t x t''))
  | Tgen (<<a>>t'') -> Tgen (<<a>>(subst t x t''))
  | Tapp (t1, t2) -> Tapp (subst t x t1, subst t x t2)
  | Tspec (t'', ty) -> Tspec (subst t x t'', ty);;
```

We can see the abstraction pattern-matches in the `Tlam` and `Tgen` cases: they are written $\texttt{<<}pat\texttt{>>}pat'$ for sub-patterns pat and pat'. But what are these magic patterns actually doing, and why should this function implement *capture-avoiding* substitution? To understand this, first suppose that the function `subst` is presented with arguments arising from the following function application:

[5] Really *atom* clashes, in fact.

```
subst (Tvar y) x (Tlam (TYvar alpha, <<y>>(Tapp (Tvar x, Tvar y))));;
```

The result of this should correspond to the open term $(\lambda y\ :\ \alpha.\ x\ y)[y/x]$, that is $\lambda z\ :\ \alpha.\ y\ z$. The Fresh O'Caml system responds as follows, which provides sufficient information to check that no variable capture has occurred.

```
Tlam (TYvar name_0, <<name_2>>(Tapp (Tvar name_1, Tvar name_2)))
```

We now explain how the system has come up with this answer. When the function `subst` is invoked as above, the second clause of the pattern-match succeeds (against a data value with constructor `Tlam`). However, before the body of the match clause is invoked then the system ensures that the atom in binding position (mapped to by the identifier `x` and occurring throughout `t''`) is *sufficiently fresh* for the current environment. In the current implementation[6], an unused atom is chosen every time such a match is taken. This will be assigned to the identifier `x` and then the term `t''` will be constructed from the original body of the abstraction value (contained within `t'`) in such a way that the new atom (call it a') appears in all of the positions where the original atom (call it a) did.

In order to perform the change of atoms we use an operation of *swapping*, otherwise known as *transposition*, to exchange a and a'. Later in this thesis we will see why swapping, rather than just renaming, is used.

The upshot of all of this is that the Barendregt variable convention is enforced across abstraction pattern-matches: when the programmer gets hold of the innards of an abstraction value after such a match then it is guaranteed that any atoms in binding position inside will be suitably fresh. This in turn implies that name clashes will not occur, so long as the programmer has correctly indicated object-level binding in their `type` declarations.

Another illuminating example is to consider two functions which attempt (somewhat inefficiently!) to calculate the free and bound variables of a PLC term. We assume the presence of a function

```
remove : 'a -> 'a list -> 'a list
```

which removes an element from a list.

```
let rec free_vars t =
  match t with
    Tvar x -> [x]
  | Tlam (ty, <<x>>t') -> remove x (free_vars t')
  | Tgen (<<a>>t') -> free_vars t'
  | Tapp (t1, t2) -> (free_vars t1) @ (free_vars t2)
  | Tspec (t', ty) -> free_vars t';;

let rec bound_vars t =
  match t with
    Tvar x -> []
  | Tlam (ty, <<x>>t') -> x :: (bound_vars t')
  | Tgen (<<a>>t') -> bound_vars t'
  | Tapp (t1, t2) -> (bound_vars t1) @ (bound_vars t2)
  | Tspec (t', ty) -> bound_vars t';;
```

Both of these functions are typeable in Fresh O'Caml, and the first is indeed a bona-fide function to calculate the free variables of a term. However, the second function should make the astute reader slightly uneasy. Indeed, we claim that it should be impossible to write a function which really does 'calculate the bound variables of a PLC term'. This should not be possible because Fresh O'Caml's correctness property (viz. §2.2.1) prevents us from distinguishing between the encodings of representatives from the same α-equivalence class of PLC terms. The function `bound_vars` is well-defined, however, because Fresh O'Caml provides fresh atoms upon each successful abstraction pattern-match. Therefore successive applications of the `bound_vars` function (even on the same terms) will *always* return distinct lists of atoms, whereas applications of the `free_vars` function on the same term will always return the same list of atoms.

[6] See §6.6.1 for possible optimisations in this area.

```
# (bound_vars (Tlam(TYvar alpha, <<x>>(Tvar x)))) =
  (bound_vars (Tlam(TYvar alpha, <<x>>(Tvar x))));;
- : bool = false
```

This example clearly illustrates how the procedure of matching against abstraction patterns has observable side-effects (in a manner analogous to the evaluation of `ref` expressions, which create new cells in Objective Caml). The older FreshML-2000 language included strict typing constraints which meant that programs such as `bound_vars`—which allow the user to observe such effects—would be ruled out. However, this system was found to be too restrictive: we explain why in §6.8. Luckily, the old belief that these restrictions were needed to enforce the desired correctness properties turned out (after some time!) to be incorrect.

Given that Fresh O'Caml permits more general abstraction types than `<<name>>`τ, we should say a few words about pattern-matching on such values. To do this we must introduce the notion of the *algebraic support* of a value, which captures the idea of 'the atoms involved in the value's construction'. Formally, it is an approximation to the *least finite support* of the denotation of the value—a notion we will introduce in Chapter 4. Indeed, the algebraic support of some value and the least finite support of its denotation will coincide so long as the value is comparable. If it is not, the algebraic support will be no smaller than the least finite support of the value's denotation.

Intuitively, the algebraic support of some value will correspond to the *free variables* of the object language term which is encodes. It is calculated by a simple structural recursion: for example, the algebraic support of an atom a is just the singleton set $\{a\}$ whilst the algebraic support of a pair `(`v`,` v'`)` is the union of the algebraic supports of v and v'. Given an abstraction value `<<`v`>>`v' then the algebraic support is calculated as the algebraic support of v' minus the algebraic support of v. In the case of function values, we have to make an approximation since the least finite support of the denotation of such a value is formally undecidable[7]: it suffices to take the union of the algebraic supports of the free variables of the function. Note that even though there is contravariance present when calculating the algebraic support of an abstraction value, this preserves the approximation since only comparable values are permitted in binding position.

Returning to the pattern-matching algorithm, consider a value `<<`v`>>`v' of type `<<`τ`>>`τ' which we wish to match against a pattern `<<`x`>>`x'. The identifier x' should be assigned the value produced from v' by freshening up any atoms therein which also occur in the algebraic support of v. Having generated sufficiently many fresh atoms, this new value is generated by a simple structural recursion over the structure of v', assigning fresh atoms at the correct points by means of swappings. The same swappings are used throughout the value v to generate the value mapped to x.

Fresh O'Caml also allowes *nested* abstraction patterns. For example, we could match against a value of type `<<name>>(<<name>>`τ`)` using a pattern of the form `<<`x`>>(<<`x'`>>`x''`)`. We shall not delve here into the exact algorithm used to decompose such complex matches, but defer the discussion until §6.5. From the programmer's point of view, the results are as expected: each abstraction is freshened up accordingly.

We can now understand the important property[8] that *abstraction values are easy to construct, but harder to deconstruct*. From an implementation point of view, we may also read 'fast' for 'easy' as we shall see later in this thesis (Chapter 6).

2.2.4 Abstraction values: two sides of the same coin

We have now seen that abstraction values do indeed provide a form of encapsulation and prevent potentially unsafe deconstruction. However at the same time they do have *concreteness* properties. For when we pattern-match against an abstraction value, we obtain value identifiers mapping to specific (carefully-chosen!) atoms with which to work. From practical experience, this delicate combination of abstractness and concreteness provides the best of both sides of the coin.

---

[7] We state exactly why in §6.8.1, where we also hit the same problem.

[8] For the purposes of this statement, equality testing on abstraction values classes as a deconstruction. We discuss this in §2.2.5.

This is exemplified by the fact that the evaluation of an abstraction expression <<e>>e' follows exactly the same pattern as a pair expression `(`e`,` e'`)`. Both e and e' are evaluated in the current environment and simply paired together to form the corresponding value.

2.2.5 Abstraction values: equality testing

Given two abstraction values <<v_1>>v_2 and <<v_1'>>v_2', we need to be able to determine when these two values are (structurally) equal. This is not an entirely trivial question to answer so we start with an example and some motivation.

We have already seen that the contextual equivalence classes of the values of types such as `term` can be proved to coincide with the α-equivalence classes of object level terms. It follows that given the values `t_1` through `t_3` from §2.2.1 then we can check whether they represent α-equivalent PLC terms just by testing them for equality in Fresh O'Caml:

```
# t1 = t2;;
- : bool = false
# t1 = t3;;
- : bool = false
# t2 = t3;;
- : bool = true
```

Very neat. However, what we are really seeing is an illusion which Fresh O'Caml presents to the programmer. At some particular point in time, a particular abstraction value will be some particular representative of the corresponding α-equivalence class (of values identified up to renaming of atoms in binding position). However, *which particular* representative exists at any point in time may only be determined by the runtime system and not by the programmer. It is thus necessary to take great care in this discussion between meta-level equivalence as exposed to the programmer and 'identity' of values as seen by the runtime system. After all, here we are defining how to manufacture the test for meta-level equivalence given the lower-level operation of a simple structural comparison[9] on values. Therefore, during the following discussion we reserve use of the word *equality* for that operation invoked by = in Fresh O'Caml. We shall use the word *identity* for no-frills structural identity on values—the testing of values for equivalence when *not* working up to renaming of atoms in binding position. In particular, a value <<v_1>>v_2 is identical to another <<v_1'>>v_2' just when v_1 is identical to v_1' and v_2 is identical to v_2'.

The process of testing two abstraction values for equality attempts to determine if the two values belong to the same equivalence class of values modulo α-conversion of atoms in binding position. In order to do this, we determine if there exists a pair of swappings of these atoms with fresh ones which respectively map the bodies of each abstraction onto two structurally identical values. Which representative of the equivalence class we end up with is irrelevant: what matters is whether such mappings exist or not. The problem is rather reminiscent of unification.

To see the algorithm in full detail, let us consider the case where the value in binding position is a single atom. Say the abstraction values to be compared are <<a_1>>v_1 and <<a_2>>v_2, where a_1 and a_2 are atoms and v_1 and v_2 are values of the same type. In order to test for equality, the types of the two abstraction values should match: indeed these two values are both of type <<`name`>>τ, for some type of bindable names `name` and where v_1 and v_2 are both of type τ. Choosing a fresh atom a_3, we can produce two swappings (permutations) of atoms: π_1 will swap a_1 and a_3; π_2 will swap a_2 and a_3. Then, the original two abstraction values will be equal iff the value obtained by recursively applying the permutation π_1 throughout the value v_1 is structurally identical to that produced by applying π_2 throughout v_2. Morally speaking, we are comparing the whole of the values minus the positions in which bound atoms occur: since in practice we have to compare the whole values then we force comparisons to succeed at these points by ensuring that the same (fresh) atoms occur at such positions in both values.

So far we have only considered abstraction values binding a single atom. Suppose instead we wish to test two values <<v_1>>v_2 and <<v_1'>>v_2' for equality, where v_1 and v_1' are of the same complicated type. In this case, we traverse the structure of the values v_1 and v_1' *in the same order for each* to yield their algebraic supports as lists $[a_1, \ldots, a_n]$ and $[a_1', \ldots, a_n']$ respectively.

[9] Think `memcmp` if you like C.

We then require the lengths of these lists to be equal. Then we allocate as many fresh atoms as a list $[c_1, \ldots, c_n]$ as there are in the support of v_1. Next, we swap each a_i with c_i throughout v_1 and see if the resulting value is identical to that obtained by swapping each a'_i with c_i throughout v'_1. Assuming this is the case, the abstraction values are deemed equal if swapping each a_i with c_i throughout v_2 yields a value identical to that obtained by swapping each a'_i with c_i throughout v'_2.

It is necessary to traverse the values v_1 and v'_1 in the same order and to then use a list (rather than a set) to store the result so that we correctly capture the positions of the various atoms throughout the syntax trees. For example, consider testing `<<[`a, b`]>>(`a`, `b`)` against `<<[`b, a`]>>(`b`, `a`)`. These values are indeed equal. The algebraic supports of the two values in binding position must be enumerated and stored in order as `[`a, b`]` and `[`b, a`]` (rather than say `[`a, b`]` and `[`a, b`]`) so that the swappings correctly lead to equal values.

As we noted before, a consequence of this algorithm is that any type in binding position (inside an abstraction type `<<`τ`>>`τ') must be one whose values are comparable. For example, attempting to test equality on values of type `<<`$\tau \to \tau'$`>>`τ'' will fail with a runtime `Invalid_argument` exception.

2.3 Explicit atom-swapping

The linearity constraint on O'Caml patterns which forbids multiple occurrences of the same variable within a single pattern means that it is not possible to write a match of the form `match t with (<<a>>x, <<a>>y) -> ...` where multiple abstractions are deconstructed using the same fresh atoms. Not only are such pattern-matches required in order to encode certain bijections which hold in the underlying FM-set theory (for example between `<<'a name>>'b * <<'a name>>'c` and `<<'a name>>('b * 'c)`), but they have been found to be necessary during practical work[21] with the language. To simulate the effect of such non-linear patterns, Fresh O'Caml provides an expression which explicitly swaps all occurrences of two atoms inside a given value. This allows the simulation of a non-linear matching as follows, using the above isomorphism as an example:

```
let f t = match t with (<<a>>x, <<b>>y) -> <<a>>(x, swap a and b in y);;
```

2.4 Fresh-for test

The boolean-valued language construct e `freshfor` e' is true iff e is of a type of names and evaluates to an atom which does not occur in the algebraic support of the value to which e' evaluates. Recalling that the algebraic support of a Fresh O'Caml representation of some object language term corresponds to the set of free names in that term, we see that the fresh-for test provides a built-in function to calculate such free names.

2.5 Something missing?

Readers who are familiar with previous work using permutation models of name binding[49, 20, 18] may be wondering where the notion of *concretion* has got to. Concretion is the action of taking an abstraction value[10] together with an atom fresh for the body and returning the body with this atom replacing the existing atom in binding position throughout. (This will be formalised in due course—see Lemma 4.2.28.) We write $v \mathbin{@} a$ to mean the concretion of the value v at the atom a.

For example, the abstraction value `<<`a`>>(`a`, `b`)` (where a, b are atoms) may be concreted at an atom $c \neq a$, b to yield the result `(`c`, `b`)`. The atom which we pick must not occur in the body in order to guarantee that no name clashes will occur. It is in fact the case that the concretion $v \mathbin{@} a$ of an abstraction $v =$ `<<`a'`>>`v' (with a single atom abstracted) at the atom a is given by the result of swapping all occurrences of a and a' throughout v'. It is therefore superfluous to incorporate both concretion and explicit atom-swapping; we have chosen the more general explicit swapping.

[10] For simplicity, we consider this to have only a single atom abstracted.

2.6 Reference cells

The interactions between reference cells and the model of name binding employed by Fresh O'Caml may be surprising to the reader. For the address of a reference cell is treated as being *pure*—that is to say, it is treated as containing no atoms. It follows that any swapping operations on the address of the cell will not have any effect—in particular, *the contents of the cell are unchanged*.

We can illustrate the semantics with the following code fragment, which always produces the result `false`[11]:

```
let a = fresh;;
let x = <<a>>(ref a);;
match x with <<a'>>a'' -> a' = !a'';;
```

Compare this to the following fragment, which always produces `true`:

```
let a = fresh;;
let x = <<a>>a;;
match x with <<a'>>a'' -> a' = a'';;
```

This semantics for swapping operations on reference cells is theoretically straightforward, but perhaps pragmatically unsatisfactory. In particular, it is likely to hinder the implementation of efficient algorithms based on the use of mutable cells. For even if a data structure only employs references for efficiency (as opposed to creating cyclic graphs, for example) then we will still hit the problem that abstraction values do not appear to 'bind' through the references. A specific example which could hit this problem would be the usual techniques of destructive unification which are employed when implementing efficient type inference algorithms. (Here, we might implement type variables as reference cells which may either contain a pointer to the type unified with that variable, or alternatively a bona-fide type variable identifier.) Whilst one could argue that the current treatment of references is morally the correct one[12], we feel that from a pragmatic perspective it is not the best thing to be doing. However, any solution is not straightforward: as we shall see in §7.1.4 it would not be correct just to allow abstractions to bind unchecked through reference cells.

2.7 A full example

As a more complete example of the use of Fresh O'Caml, Figure 2.2 presents a complete type-checker for the polymorphic lambda calculus. We hope that the reader agrees that the code has a certain elegance about it.

2.8 Pretty-printing

We have seen how Fresh O'Caml allows access to concrete names. This gives rise to the problem that a very limited set of operations are provided to manipulate such names. In particular, there appears to be no straightforward way to somehow tag a name with a textual string—as one would want to do in order to pretty-print syntax trees or to report error messages, for example. Many potential applications of Fresh O'Caml (compilers, theorem provers, etc.) receive user input and it is obviously desirable to maintain as many of the user's textual names as possible. (We might extend them with numeric tags: this might be necessary if fresh atoms representing the name have been generated to prevent a capture.)

At first it might seem that the correct way forward is to represent object language names by (atom, string) pairs. However this soon breaks down because the string is unaffected by any swappings applied to the pair, meaning that the correspondence can become garbled. James Leifer at INRIA Rocquencourt pointed out to us that the solution is to instead tag the *binding points* in the object language syntax trees with the original textual names. These will correspond to those nodes in the syntax trees where abstraction types feature. When such a

[11] As of the time of writing (February 2005), there is a bug in the released versions of the Fresh O'Caml system which causes incorrect results when using reference cells. This will be fixed in the near future.

[12] References are, after all, part of the 'awkward squad' as Peyton-Jones puts it[40].

```
type t and var = t name;;
type t and tyvar = t name;;

type typ = TYvar of tyvar
         | TYfn of typ * typ
         | TYforall of <<tyvar>>typ;;

type term = Tvar of var
          | Tlam of typ * (<<var>>term)
          | Tgen of <<tyvar>>term
          | Tapp of term * term
          | Tspec of term * typ;;

exception Arg_type_mismatch;;
exception Bad_application;;
exception Bad_specialisation;;
exception Not_typeable;;
exception Unbound_var;;

let rec lookup_env env x =
  match env with
    [] -> raise Unbound_var
  | ((x', ty)::_) when x = x' -> ty
  | (_::env') -> lookup_env env' x;;

let rec extend_env env x ty = (x, ty) :: env;;

let rec ty_subst ty' a ty =
  match ty with
    TYvar a' when a = a' -> ty'
  | TYvar _ -> ty
  | TYfn (ty1, ty2) -> TYfn (ty_subst ty' a ty1, ty_subst ty' a ty2)
  | TYforall (<<a'>>ty'') -> TYforall (<<a'>>(ty_subst ty' a ty''));;

let rec typecheck' env t =
  match t with
    Tvar x -> lookup_env env x
  | Tlam (ty, <<x>>t') -> TYfn (ty, typecheck' (extend_env env x ty) t')
  | Tgen (<<a>>t') -> TYforall (<<a>>(typecheck' env t'))
  | Tapp (t1, t2) ->
      begin match (typecheck' env t1, typecheck' env t2) with
        (TYfn (arg_ty, res_ty), ty) ->
          if arg_ty = ty then res_ty else raise Arg_type_mismatch
      | _ -> raise Bad_application
      end
  | Tspec (Tgen (<<a>>t'), ty) -> ty_subst ty a (typecheck' env t')
  | Tspec _ -> raise Bad_specialisation;;

let typecheck = typecheck' [];;
```

Figure 2.2: Typechecker for PLC.

syntax tree node is deconstructed at runtime we obtain a fresh atom and can then pick a textual name for it based on the original user's name. (Names which were free in the first place may simply use the technique described in the previous paragraph, as they will be unaffected by the side-effects arising from the deconstruction of abstraction values.) We now present a further enhancement to this technique. For this example we keep the object language simple and just use the terms of the untyped λ-calculus, inductively defined by the following grammar:

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

```
type t and var = t name;;
type lam = Var of var | Lam of string * <<var>>lam | App of lam * lam;;

let rec lookup_atom varmap a =
  match varmap with
    [] -> raise Not_found
  | (a', s) :: xs -> if a = a' then s else lookup_atom xs a;;

let rec name_used varmap s =
  match varmap with
    [] -> false
  | (_, s') :: xs -> if s = s' then true else name_used xs s;;

let allocate_name varmap a s =
  if name_used varmap s then let s' = s ^ "'" in (s', (a, s') :: varmap)
  else (s, (a, s) :: varmap);;

let rec print_lam varmap t =
  match t with
    Var a -> lookup_atom varmap a
  | Lam (s, <<a>>t') ->
      let cont = function (var_text, new_varmap) ->
        "fn " ^ var_text ^ " => " ^ (print_lam new_varmap t')
      in
        (* if the atom a never occurs in t', then we don't need to
           record the use of the textual name s *)
        cont (if a freshfor t' then (s, varmap) else allocate_name varmap a s)
  | App (t1, t2) -> "(" ^ (print_lam varmap t1) ^ ") " ^
                    "(" ^ (print_lam varmap t2) ^ ")";;

let print = print_lam [];;
let x, y = fresh, fresh;;
let test1 = Lam ("x", <<x>>(Lam ("x", <<x>>(Var x))));;
let test2 = Lam ("y", <<y>>(App(Lam ("x", <<x>>(Lam ("y",
                 <<y>>(App (Var x, Var y))))), Var y)));;

# print test1;;
- : string = "fn x => fn x => x"
# print test2;;
- : string = "fn y => (fn x => fn y' => (x) (y')) (y)"
```

Figure 2.3: Sample pretty-printing code, tests and output.

If the program receives input (encoded in some indicative textual format) `fun x -> fun x -> x` and subsequently manipulates the parsed term (indeed, even if it just parses it to a syntax tree and then pretty-prints it) then we would hope that only one textual name appears in the output. This ought to be `x`, of course. No second name should be chosen since the second binding of `x` shadows the first and there are no uses of the outermost `x`. How do we adapt our algorithm to cope with this? Our answer presents an elegant use of the `freshfor` construct. When encountering a binding point in the syntax tree, we take the atom which the abstraction pattern-match has provided us with and check if it is fresh for the body using `freshfor`. If it is, we know that the variable is actually unused and therefore its (newly-generated) textual name may be re-used later.

Figure 2.3 presents a sample pretty-printer for closed λ-terms incorporating these techniques, together with its output. For simplicity, we omit code to prevent superfluous parentheses arising in the output; also, we use deeply-inefficient routines for handling maps keyed on atoms. Why do we not use the standard library? The answer is that ways of incorporating names into data structures such as hash tables and balanced binary trees are still under investigation as we will eventually see in §7.1.2.

3 Mini-FreshML, operationally

'Inside every large language is a small language struggling to get out.' —Igarashi et al.[25]

IN THIS CHAPTER we turn our attention to the formalisation of the new constructs seen in Fresh O'Caml with a view to proving some observational equivalence results. We do this by defining a small language *Mini-FreshML* which provides for the key features:

- the dynamic allocation of fresh atoms;
- the creation and deconstruction of atom-abstractions;
- the process of atom-swapping;
- user-defined datatypes involving atom-abstractions.

After introducing the syntax and static semantics of Mini-FreshML, we will give a traditional 'big-step' operational semantics to describe its evaluation behaviour. This will then be augmented via an operational semantics based on *frame stacks*[48]—a particular representation of the familiar *evaluation*, or *reduction* contexts[29, 71]. We shall need this alternative formulation in Chapter 5 where we give a denotational semantics to Mini-FreshML. It is by using this denotational semantics that we will prove results about certain notions of observational equivalence of Mini-FreshML expressions.

3.1 Syntax

We assume a countably infinite set VId of *value identifiers* (typical element x) and a countably infinite set $\mathbb{A}$ of *atoms* (typical element a) disjoint from VId. The expressions and canonical forms of Mini-FreshML are then the abstract syntax trees generated by the grammars in Figure 3.1. The unusual language constructs have been highlighted like this. Since we will be working with a 'substituted-in' style of semantics (as opposed to 'environment style', such as in [36, 63] and discussed in §3.5) then the canonical forms (also known as *values*) are a subset of the expressions.

Write Exp_τ and Val_τ for the closed expressions and canonical forms of type τ respectively. Write Val for the set of *all* closed canonical forms.

An important point to note is that *expressions are to be identified up to α-conversion of bound value identifiers*. The binding forms are as follows, with the binding positions underlined.

```
fun x(x) = [–]          match e with
let x = e in [–]          C1(x1) -> [–]
let (x, x) = e in [–]   | ···
let <<x>>x = e in [–]   | CK(xK) -> [–]
```

Note that the atom-abstraction expression- and value-former <<–>>– is *not a binder*. However, properties of Mini-FreshML observational equivalence will turn out to have the following consequences:

- For distinct atoms a, a' then the value <<a>>a is observationally equivalent to <<a'>>a'.
- For distinct value identifiers x, x' the value <<x>>x is observationally equivalent to <<x'>>x'.
- For distinct value identifiers x, y and z the value <<y>>x is not observationally equivalent to <<z>>x.

3.2 Static semantics

Mini-FreshML is a strongly-typed language, although we use a monomorphic type system in order to keep the presentation as simple as possible. (Therefore, in particular, `let`-expressions do not introduce polymorphism unlike in Fresh O'Caml). Expressions are assigned types generated by the grammar in Figure 3.2; let us write Typ for the set of all such types and assume the presence of a single top-level type δ. Values of this type may be built using the data constructors $\mathtt{C}_1, \cdots, \mathtt{C}_K$, which have argument types $\sigma_1, \cdots, \sigma_K$ respectively. The σ_k are built from the same grammar as types τ (and hence may contain occurrences of δ, in particular). Therefore δ corresponds to a Fresh O'Caml `type` declaration of the form:

$$\mathtt{type}\ \delta\ \mathtt{=}\ \mathtt{C}_1\ \mathtt{of}\ \sigma_1\ \mathtt{|}\ \cdots\ \mathtt{|}\ \mathtt{C}_K\ \mathtt{of}\ \sigma_K.$$

The theory which follows could be easily extended to cope with multiple, possibly mutually-recursive top-level datatype declarations. We write Γ for finite maps from value identifiers to types, thought of as *typing contexts*; expressions may be assigned types in such contexts using the *typing relation* $\vdash$. This is a set of triples (Γ, e, τ) where Γ is a typing context, e is an expression and τ is a type. We write $\Gamma \vdash e\ :\ \tau$ iff (Γ, e, τ) is in the relation, which is defined by structural recursion on e according to the axioms and rules in Figure 3.3. Write $\Gamma, x\ :\ \tau$ to indicate that typing context mapping x to τ and otherwise acting as Γ. In the event that Γ is empty (and hence e is closed), we abbreviate the notation and write $\vdash e\ :\ \tau$. Since the canonical forms are a subset of the expressions, then they are also assigned types according to Figure 3.3.

Definition 3.2.1 (Atom sets and atom swapping). Given an expression e, let $\mathrm{atms}(e)$ be the finite set of all atoms occurring within e. Given atoms a, a', write $(a\ a') \cdot e$ for that expression formed by recursively exchanging all occurrences of a and a' throughout the expression e.

---

e	::=	x	value identifier
	|	`()`	unit
	|	a	atom
	|	$\mathtt{C}_k(e)$	data construction
	|	`(`e`,` e`)`	pairing
	|	`fresh`	fresh name creation
	|	`<<`e`>>`e	abstraction
	|	`swap` e`,` e `in` e	atom-swapping
	|	`fun` x`(`x`)` `=` e	recursive function
	|	$e\ e$	function application
	|	`if` e `=` e `then` e `else` e	conditional
	|	`let` x `=` e `in` e	value binding
	|	`let (`x`,` x`)` `=` e `in` e	pair deconstruction
	|	`let <<`x`>>`x `=` e `in` e	abstraction deconstruction
	|	`match` e `with`	data value deconstruction
		$\mathtt{C}_1(x_1)$ `->` e_1	
	|	$\cdots$	
	|	$\mathtt{C}_K(x_K)$ `->` e_K	

v	::=	x	value identifier
	|	`()`	unit
	|	a	atom
	|	$\mathtt{C}_k(v)$	data construction
	|	`(`v`,` v`)`	pairing
	|	`<<`v`>>`v	abstraction
	|	`fun` x`(`x`)` `=` e	recursive function

Figure 3.1: Expressions and canonical forms of Mini-FreshML.

τ	$::=$	`unit`	unit type
	$\mid$	`name`	name type
	$\mid$	δ	data type
	$\mid$	$\tau \times \tau$	pair type
	$\mid$	`<<name>>`τ	abstraction type
	$\mid$	$\tau \to \tau$	function type

Figure 3.2: Types of Mini-FreshML.

Given a permutation π, write $\pi \cdot e$ for that expression formed by recursively applying the permutation π to the atoms throughout e. $\diamond$

Definition 3.2.2 (Substitutions). Let *substitutions* ψ be finite maps from value identifiers to closed canonical forms. Write $e[\psi]$ for that expression formed from another expression e by the capture-avoiding application of the substitution ψ. Given a typing context Γ, write $\vdash \psi : \Gamma$ just when $\mathrm{dom}(\psi) = \mathrm{dom}(\Gamma)$ and for each $x \in \mathrm{dom}(\psi)$, $\vdash \psi(x) : \Gamma(x)$. Write $\psi, x \mapsto v$ for that substitution mapping x to v and otherwise behaving as ψ. Occasionally for clarity we will use the notation $\psi[x \mapsto v]$ instead of $\psi, x \mapsto v$. $\diamond$

The construction of the typing rules immediately gives the following two lemmata, whose proof follows by simple inspection.

Lemma 3.2.3 (Substitutivity). *For substitutions ψ, typing contexts Γ and expressions e then $\vdash \psi : \Gamma$ implies that $\Gamma \vdash e : \tau \Leftrightarrow \vdash e[\psi] : \tau$.* $\square$

Lemma 3.2.4 (Equivariance for typing judgements). *For any expression e in context Γ and atoms a, a' then $\Gamma \vdash e : \tau \Leftrightarrow \Gamma \vdash (a\ a') \cdot e : \tau$.* $\square$

The simplicity of the typing rules may be surprising (especially to those familiar with [49]), but no further complication is required. This means that production of a Hindley-Milner style type inference algorithm[1] for inferring the types of our language constructs is straightforward. For example, note the similarity between the rules for abstraction and pair expressions. This is in contrast with the original FreshML-2000 design which made use of a complicated static type system (discussed at length in §6.8). The lack of such a type system is particularly helpful when considering how to modify existing compilers in order to support our features, such as we have done for Objective Caml.

Let us now move on and consider the evaluation behaviour of Mini-FreshML, which presents more novelties than the static semantics.

3.3 Dynamic semantics via big-step

In this section we provide a big-step operational semantics which defines how expressions evaluate to canonical forms. Overall the evaluation strategy agrees with that of Fresh O'Caml because Mini-FreshML is also a strict call-by-value language. The main point to bear in mind is that during the evaluation of expressions we may have the computational effects arising from the allocation of fresh atoms. This happens not only via the `fresh` expression but also by pattern-matching against abstraction patterns by using `let <<`x`>>`x' `=` e `in` e'.

In order to cope with this, we provide an evaluation relation on 4-tuples $(\overline{a}, e, v, \overline{a}')$ where $\overline{a}$ and $\overline{a}'$ are finite sets of atoms, e is a closed expression and v a closed canonical form. We write $\overline{a}, e \Downarrow v, \overline{a}'$ iff $(\overline{a}, e, v, \overline{a}')$ is in the relation. Such a judgement is read as

> *In the state where the atoms $\overline{a}$ have been allocated, the expression e evaluates to the value v and allocates the atoms $\overline{a}' \setminus \overline{a}$.*

The evaluation relation is inductively defined by the axioms and rules in Figure 3.4, but restricted to those judgements $\overline{a}, e \Downarrow v, \overline{a}'$ when $\overline{a}' \supseteq \overline{a}$. From inspection of that Figure, we obtain the following three lemmata and then a *subject reduction* result.

[1] Whose roots may lie as far back as the work of Curry in the 1930s, or possibly even Tarski in the 1920s.

$$\text{vid}\,\frac{}{\Gamma \vdash x \,:\, \tau}\; x \in \mathrm{dom}(\Gamma) \text{ and } \tau = \Gamma(x)$$

$$\text{atom}\,\frac{}{\Gamma \vdash a \,:\, \texttt{name}}\; a \in \mathbb{A} \qquad \text{unit}\,\frac{}{\Gamma \vdash \texttt{()} \,:\, \texttt{unit}}$$

$$\text{con}\,\frac{\Gamma \vdash e \,:\, \sigma_k}{\Gamma \vdash \mathtt{C}_k\texttt{(}e\texttt{)} \,:\, \delta} \qquad \text{fresh}\,\frac{}{\Gamma \vdash \texttt{fresh} \,:\, \texttt{name}}$$

$$\text{pair}\,\frac{\Gamma \vdash e \,:\, \tau \quad \Gamma \vdash e' \,:\, \tau'}{\Gamma \vdash \texttt{(}e\texttt{, }e'\texttt{)} \,:\, \tau \times \tau'} \qquad \text{abst}\,\frac{\Gamma \vdash e \,:\, \texttt{name} \quad \Gamma \vdash e' \,:\, \tau}{\Gamma \vdash \texttt{<<}e\texttt{>>}e' \,:\, \texttt{<<name>>}\tau}$$

$$\text{swap}\,\frac{\Gamma \vdash e_1 \,:\, \texttt{name} \quad \Gamma \vdash e_2 \,:\, \texttt{name} \quad \Gamma \vdash e_3 \,:\, \tau}{\Gamma \vdash \texttt{swap } e_1\texttt{, } e_2 \texttt{ in } e_3 \,:\, \tau}$$

$$\text{fun}\,\frac{\Gamma, f \,:\, \tau \to \tau', x \,:\, \tau \vdash e \,:\, \tau'}{\Gamma \vdash \texttt{fun } f\texttt{(}x\texttt{) = } e \,:\, \tau \to \tau'} \qquad \text{app}\,\frac{\Gamma \vdash e \,:\, \tau' \to \tau \quad \Gamma \vdash e' \,:\, \tau'}{\Gamma \vdash e\; e' \,:\, \tau}$$

$$\text{let}\,\frac{\Gamma \vdash e \,:\, \tau' \quad \Gamma, x \,:\, \tau' \vdash e' \,:\, \tau}{\Gamma \vdash \texttt{let } x \texttt{ = } e \texttt{ in } e' \,:\, \tau}$$

$$\text{let-pair}\,\frac{\Gamma \vdash e \,:\, \tau_1 \times \tau_2 \quad \Gamma, x \,:\, \tau_1, x' \,:\, \tau_2 \vdash e' \,:\, \tau}{\Gamma \vdash \texttt{let (}x\texttt{, }x'\texttt{) = } e \texttt{ in } e' \,:\, \tau}$$

$$\text{let-abst}\,\frac{\Gamma \vdash e \,:\, \texttt{<<name>>}\tau' \quad \Gamma, x \,:\, \texttt{name}, x' \,:\, \tau' \vdash e' \,:\, \tau}{\Gamma \vdash \texttt{let <<}x\texttt{>>}x' \texttt{ = } e \texttt{ in } e' \,:\, \tau}$$

$$\text{if}\,\frac{\Gamma \vdash e \,:\, \texttt{name} \quad \Gamma \vdash e' \,:\, \texttt{name} \quad \Gamma \vdash e_1 \,:\, \tau \quad \Gamma \vdash e_2 \,:\, \tau}{\Gamma \vdash \texttt{if } e \texttt{ = } e' \texttt{ then } e_1 \texttt{ else } e_2 \,:\, \tau}$$

$$\text{match}\,\frac{\Gamma \vdash e \,:\, \delta \quad \text{for all } 1 \le k \le K, \Gamma, x_k \,:\, \sigma_k \vdash e_k \,:\, \tau}{\Gamma \vdash \texttt{match } e \texttt{ with } \cdots \texttt{| } \mathtt{C}_k\texttt{(}x_k\texttt{) -> } e_k \texttt{ | } \cdots \,:\, \tau}$$

Figure 3.3: Typing rules for expressions.

Lemma 3.3.1 (State growth during evaluation). *Given an expression e and a set of atoms $\overline{a}$ containing the atoms of e, then $\overline{a}, e \Downarrow v, \overline{a}'$ implies that $\overline{a}' \supseteq \overline{a}$ and $\mathrm{atms}(v) \subseteq \overline{a}'$.* □

Lemma 3.3.2 (Equivariance for the big-step relation). *For any atoms a, a' then $\overline{a}, e \Downarrow v, \overline{a}'$ implies that $(a\ a') \cdot \overline{a}, (a\ a') \cdot e \Downarrow (a\ a') \cdot v, (a\ a') \cdot \overline{a}'$, where $(a\ a') \cdot -$ acting on a finite set of atoms $\overline{a}$ is that finite set given by exchanging all occurrences of a and a' throughout $\overline{a}$.* □

Lemma 3.3.3 (Determinacy). *If $\overline{a}, e \Downarrow v, \overline{a}'$ and $\overline{a}, e \Downarrow v', \overline{a}''$ then there exists a permutation $\pi \,:\, (\overline{a}' \setminus \overline{a}) \cong (\overline{a}'' \setminus \overline{a})$ such that $v' = \pi \cdot v$.* □

Lemma 3.3.4 (Subject reduction). *Take a substitution ψ, a typing context Γ such that $\vdash \psi \,:\, \Gamma$ and an expression e with $\Gamma \vdash e \,:\, \tau$. Then for all $\overline{a}' \supseteq \overline{a} \supseteq \mathrm{atms}(e, \psi)$, whenever there exists a derivable judgement $\overline{a}, e[\psi] \Downarrow v, \overline{a}'$ then $\vdash v \,:\, \tau$.*

Proof. By the substitutivity property of the typing relation (Lemma 3.2.3), it suffices to prove that for all ψ with $\vdash \psi \,:\, \Gamma$ and any $\overline{a} \supseteq \mathrm{atms}(e, \psi)$,

$$\forall v, \overline{a}' \supseteq \overline{a}.\ (\vdash e[\psi] \,:\, \tau \ \wedge\ \overline{a}, e[\psi] \Downarrow v, \overline{a}') \Rightarrow \vdash v \,:\, \tau$$

by induction over the axioms and rules defining the evaluation relation $\Downarrow$. We give just those cases which are specific to Mini-FreshML.

► *Case (fresh).* Since $\texttt{fresh}[\psi] = \texttt{fresh}$ then the result follows immediately by observing the (fresh) rule and noting that for any $a \in \mathbb{A}$, $\vdash a \,:\, \texttt{name}$.

► *Case (abst).* First note that $(\texttt{<<}e\texttt{>>}e')[\psi] = \texttt{<<}e[\psi]\texttt{>>}e'[\psi]$. Without loss of generality we have that $\vdash \texttt{<<}e[\psi]\texttt{>>}e'[\psi] \,:\, \texttt{<<name>>}\tau$; the typing rules then tell us that $\vdash e[\psi] \,:\, \texttt{name}$ and $\vdash e'[\psi] \,:\, \tau$. Take any $\overline{a} \supseteq \mathrm{atms}(\texttt{<<}e[\psi]\texttt{>>}e'[\psi])$ and $\overline{a}'' \supseteq \overline{a}$; then $\mathrm{atms}(e[\psi]) \subseteq \overline{a}$ and

$$\text{atom}\frac{}{\overline{a}, a \Downarrow a, \overline{a}}\ a \in \overline{a} \qquad \text{unit}\frac{}{\overline{a}, () \Downarrow (), \overline{a}}$$

$$\text{con}\frac{\overline{a}, e \Downarrow v, \overline{a}'}{\overline{a}, \mathtt{C}_k(e) \Downarrow \mathtt{C}_k(v), \overline{a}'} \qquad \text{fresh}\frac{}{\overline{a}, \mathtt{fresh} \Downarrow a, \overline{a} \uplus \{a\}}\ a \in \mathbb{A} \setminus \overline{a}$$

$$\text{pair}\frac{\overline{a}, e \Downarrow v, \overline{a}' \qquad \overline{a}', e' \Downarrow v', \overline{a}''}{\overline{a}, (e,\ e') \Downarrow (v,\ v'), \overline{a}''} \qquad \text{abst}\frac{\overline{a}, e \Downarrow v, \overline{a}' \qquad \overline{a}', e' \Downarrow v', \overline{a}''}{\overline{a}, \texttt{<<}e\texttt{>>}e' \Downarrow \texttt{<<}v\texttt{>>}v', \overline{a}''}$$

$$\text{swap}\frac{\overline{a}, e \Downarrow a, \overline{a}' \qquad \overline{a}', e' \Downarrow a', \overline{a}'' \qquad \overline{a}'', e'' \Downarrow v, \overline{a}'''}{\overline{a}, \mathtt{swap}\ e,\ e'\ \mathtt{in}\ e'' \Downarrow (a\ a') \cdot v, \overline{a}'''}$$

$$\text{fun}\frac{}{\overline{a}, \mathtt{fun}\ f(x)\ =\ e \Downarrow \mathtt{fun}\ f(x)\ =\ e, \overline{a}}$$

$$\text{app}\frac{\begin{array}{c}\overline{a}, e \Downarrow \mathtt{fun}\ f(x)\ =\ e'', \overline{a}' \qquad \overline{a}', e' \Downarrow v', \overline{a}'' \\ \overline{a}'', e''[\mathtt{fun}\ f(x)\ =\ e''/f, v'/x] \Downarrow v, \overline{a}'''\end{array}}{\overline{a}, e\ e' \Downarrow v, \overline{a}'''}$$

$$\text{let}\frac{\overline{a}, e \Downarrow v', \overline{a}' \qquad \overline{a}', e'[v'/x] \Downarrow v, \overline{a}''}{\overline{a}, \mathtt{let}\ x\ =\ e\ \mathtt{in}\ e' \Downarrow v, \overline{a}''}$$

$$\text{let-pair}\frac{\overline{a}, e \Downarrow (v',\ v''), \overline{a}' \qquad \overline{a}', e'[v'/x, v''/x'] \Downarrow v, \overline{a}''}{\overline{a}, \mathtt{let}\ (x,\ x')\ =\ e\ \mathtt{in}\ e' \Downarrow v, \overline{a}''}$$

$$\text{let-abst}\frac{\overline{a}, e \Downarrow \texttt{<<}a\texttt{>>}v', \overline{a}' \qquad \overline{a}' \uplus \{a'\}, e'[a'/x, (a\ a') \cdot v'/x'] \Downarrow v, \overline{a}''}{\overline{a}, \mathtt{let}\ \texttt{<<}x\texttt{>>}x'\ =\ e\ \mathtt{in}\ e' \Downarrow v, \overline{a}''}\ a' \in \mathbb{A} \setminus \overline{a}'$$

$$\text{if}\frac{\begin{array}{c}\overline{a}, e \Downarrow a, \overline{a}' \qquad \overline{a}', e' \Downarrow a', \overline{a}'' \\ a = a' \Rightarrow \overline{a}'', e_1 \Downarrow v, \overline{a}''' \qquad a \neq a' \Rightarrow \overline{a}'', e_2 \Downarrow v, \overline{a}'''\end{array}}{\overline{a}, \mathtt{if}\ e\ =\ e'\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \Downarrow v, \overline{a}'''}$$

$$\text{match}\frac{\overline{a}, e \Downarrow \mathtt{C}_j(v_j), \overline{a}' \wedge \overline{a}', e_j[v_j/x_j] \Downarrow v, \overline{a}''}{\overline{a}, \mathtt{match}\ e\ \mathtt{with}\ \cdots \mid \mathtt{C}_k(x_k)\ \texttt{->}\ e_k\ \mid\ \cdots \Downarrow v, \overline{a}''}$$

Figure 3.4: Big-step semantics.

$\mathrm{atms}(e'[\psi]) \subseteq \overline{a}$. Now if $\overline{a}, \texttt{<<}e[\psi]\texttt{>>}e'[\psi] \Downarrow \texttt{<<}v\texttt{>>}v', \overline{a}''$, then by Lemma 3.3.1 there exists $\overline{a}'$ with $\overline{a}'' \supseteq \overline{a}' \supseteq \overline{a}$ such that $\overline{a}, e[\psi] \Downarrow v, \overline{a}'$ and $\overline{a}', e'[\psi] \Downarrow v', \overline{a}''$. Therefore, $\mathrm{atms}(e'[\psi]) \subseteq \overline{a}'$. By applying the induction hypotheses we can then deduce that $\vdash v : \mathtt{name}$ and $\vdash v' : \tau$. It follows that $\vdash \texttt{<<}v\texttt{>>}v' : \texttt{<<name>>}\tau$.

► *Case (swap).* Take atom-sets $\overline{a}''' \supseteq \overline{a} \supseteq \mathrm{atms}(\mathtt{swap}\ e[\psi],\ e'[\psi]\ \mathtt{in}\ e''[\psi])$; now assume that $\vdash \mathtt{swap}\ e[\psi],\ e'[\psi]\ \mathtt{in}\ e''[\psi] : \tau$ and $\overline{a}, \mathtt{swap}\ e[\psi],\ e'[\psi]\ \mathtt{in}\ e''[\psi] \Downarrow v, \overline{a}'''$. Then Lemma 3.3.1 implies that there exist further sets of atoms $\overline{a}'$, $\overline{a}''$ satisfying the properties $\overline{a} \subseteq \overline{a}' \subseteq \overline{a}'' \subseteq \overline{a}'''$, $\mathrm{atms}(e[\psi]) \subseteq \overline{a}$, $\mathrm{atms}(e'[\psi]) \subseteq \overline{a}'$ and $\mathrm{atms}(e''[\psi]) \subseteq \overline{a}''$. By the induction hypotheses, these come along with values a, a' and v satisfying $\vdash a : \mathtt{name}$, $\vdash a' : \mathtt{name}$ and $\vdash v : \tau$. Now we apply Lemma 3.2.4 to get that $\vdash (a\ a') \cdot v : \tau$.

► *Case (let-abst).* Consider $\overline{a}'' \supseteq \overline{a} \supseteq \mathrm{atms}(\mathtt{let}\ \texttt{<<}x\texttt{>>}x' = e[\psi]\ \mathtt{in}\ e'[\psi])$. Assume that we have the judgements $\vdash \mathtt{let}\ \texttt{<<}x\texttt{>>}x' = e[\psi]\ \mathtt{in}\ e'[\psi] : \tau$ and $\mathtt{let}\ \texttt{<<}x\texttt{>>}x' = e[\psi]\ \mathtt{in}\ e'[\psi] \Downarrow v, \overline{a}''$. Then the typing rules together with Lemma 3.2.3 give us that $\vdash e[\psi] : \texttt{<<name>>}\tau'$ and $\vdash e'[\psi, x \mapsto a', x' \mapsto v'] : \tau$, for any atom a' and a closed v' of type τ'. One induction hypothesis together with Lemma 3.3.1 now tells us that there exists a set of atoms $\overline{a}'$ with $\overline{a}' \supseteq \overline{a}$, $\overline{a}, e[\psi] \Downarrow \texttt{<<}a\texttt{>>}v', \overline{a}'$ and $\vdash v' : \tau'$. Taking any $a' \in \mathbb{A} \setminus \overline{a}'$, Lemma 3.2.4 then tells us that $\vdash (a\ a') \cdot v' : \tau$. From earlier we now deduce that $\vdash e'[\psi, x \mapsto a', x' \mapsto (a\ a') \cdot v'] : \tau$. Since $\mathrm{atms}(e'[a'/x, (a\ a') \cdot v'/x']) \subseteq \overline{a}' \uplus \{a'\}$ then $\vdash \psi, x \mapsto a' : \Gamma, x \mapsto \mathtt{name}, x' \mapsto \tau'$. Therefore the second induction hypothesis together with Lemma 3.3.1 tells us that there exists $\overline{a}'' \supseteq \overline{a}' \uplus \{a'\}$ with $\overline{a}' \uplus \{a'\}, e'[a'/x, (a\ a') \cdot v'/x'] \Downarrow v, \overline{a}''$ and furthermore $\vdash v : \tau$. □

$\mathtt{C}_k(\mathtt{[-]})$	`if` v `= [-] then` e `else` e
`([-],` e`)`	`[-]` e
`(`v`, [-])`	v `[-]`
`<<[-]>>`e	`let` x `= [-] in` e
`<<`v`>>[-]`	`let (`x`,` x'`) = [-] in` e
`swap [-],` e `in` e	`let <<`x`>>`x' `= [-] in` e
`swap` v`, [-] in` e	`match [-] with`
`swap` v`,` v `in [-]`	$\cdots$ `|` $\mathtt{C}_k(x_k)$ `->` e_k `|` $\cdots$
`if [-] =` e `then` e `else` e	

Figure 3.5: Evaluation frames $\mathcal{F}$.

3.4 Dynamic semantics via frame stacks

In this section we introduce an operational semantics for Mini-FreshML which draws on the theory of *frame stack semantics*[48]. This has two distinct advantages over the previous big-step semantics:

- it provides a structurally-inductive definition of what it means for the evaluation of an expression to terminate;
- it enables reasoning about evaluation properties without using explicit name sets.

For our application, the second of these properties turns out to be the important one. For when developing a denotational semantics for Mini-FreshML, it is much easier to work in a world where the dynamically-allocated names are kept implicit. The domain theory developed in Chapter 4 will enable us to do this in the denotational semantics, which will match up well with this new operational model.

3.4.1 A termination relation

What we are going to do is consider the termination behaviour of expressions e in a *frame stack* S, defined as follows.

Definition 3.4.1 (Frames and frame stacks). Let an *evaluation frame* $\mathcal{F}$ be as in Figure 3.5. A *frame stack* S consists of a possibly-empty list of frames like so: $S ::= [] \mid S \circ \mathcal{F}$. Write $S \mathbin{@} S'$ for the frame stack consisting of S' appended to S. $\diamond$

Each evaluation frame acts like a fragment of an evaluation context; we can think of a frame stack as representing one whole context. Frame stacks are assigned types using a typing relation $\vdash_s$. This is a set of 4-tuples (Γ, S, τ, τ') where Γ is a typing context, S is a frame stack and τ, τ' are types. We write $\Gamma \vdash_s S : \tau \multimap \tau'$ iff (Γ, S, τ, τ') is in the relation, which is inductively defined by the axiom and rules in Figure 3.6. Informally, a frame stack of type $\tau \multimap \tau'$ accepts an argument of type τ in the hole and evaluates to a value of type τ'. Often we will not concern ourselves with the result type, simply writing types such as $\tau \multimap \_$ instead.

In Figure 3.7 we give a set of axioms and rules which define a binary relation between frame stacks and expressions known as the *termination relation*[2]. In those Figures, e, e', $\ldots$ stand for expressions, v, v', $\ldots$ stand for canonical forms and a, a', $\ldots$ stand for atoms. The definition is split into two parts according to whether we are currently considering a canonical form or a non-canonical expression. For clarity, write $\langle S, e\rangle\downarrow$ iff the pair (S, e) lies in the relation. In this case, we say that 'e terminates when evaluated in stack S'.

Note how the definition of the termination relation is both structurally inductive and also free of explicit atom-sets: the frame stacks encapsulate all the necessary information about

[2]We could consider an abstract machine to interpret Mini-FreshML terms which evaluates using frame stacks in a way corresponding to the termination relation $\langle S, e\rangle\downarrow$. However, the semantics which we give here is not directly suitable for this purpose since, in order to make transitions deterministic, it would require runtime tests to determine if arbitrary expressions were in canonical form. A possible solution to this problem would be to use a *CK/VK machine* in the style of Levy[28] and we refer the reader to this work for details.

$$\text{empty}\frac{}{\Gamma \vdash_s [] \;:\; \tau \multimap \tau} \qquad \text{con}\frac{\Gamma \vdash_s S \;:\; \delta \multimap \tau}{\Gamma \vdash_s S \circ \mathtt{C}_k([\text{–}]) \;:\; \sigma_k \multimap \tau}$$

$$\text{pair-l}\frac{\Gamma \vdash_s S \;:\; \tau_1 \times \tau_2 \multimap \tau \quad \Gamma \vdash e \;:\; \tau_2}{\Gamma \vdash_s S \circ ([\text{–}],\ e) \;:\; \tau_1 \multimap \tau}$$

$$\text{pair-r}\frac{\Gamma \vdash_s S \;:\; \tau_1 \times \tau_2 \multimap \tau \quad \Gamma \vdash v \;:\; \tau_1}{\Gamma \vdash_s S \circ (v,\ [\text{–}]) \;:\; \tau_2 \multimap \tau}$$

$$\text{abst-l}\frac{\Gamma \vdash_s S \;:\; \texttt{<<name>>}\tau \multimap \tau' \quad \Gamma \vdash e \;:\; \tau}{\Gamma \vdash_s S \circ \texttt{<<}[\text{–}]\texttt{>>}e \;:\; \texttt{name} \multimap \tau'}$$

$$\text{abst-r}\frac{\Gamma \vdash_s S \;:\; \texttt{<<name>>}\tau \multimap \tau' \quad \Gamma \vdash v \;:\; \texttt{name}}{\Gamma \vdash_s S \circ \texttt{<<}v\texttt{>>}[\text{–}] \;:\; \tau \multimap \tau'}$$

$$\text{swap-1}\frac{\Gamma \vdash_s S \;:\; \tau \multimap \tau' \quad \Gamma \vdash e' \;:\; \texttt{name} \quad \Gamma \vdash e'' \;:\; \tau}{\Gamma \vdash_s S \circ \texttt{swap}\ [\text{–}],\ e'\ \texttt{in}\ e'' \;:\; \texttt{name} \multimap \tau'}$$

$$\text{swap-2}\frac{\Gamma \vdash_s S \;:\; \tau \multimap \tau' \quad \Gamma \vdash v \;:\; \texttt{name} \quad \Gamma \vdash e'' \;:\; \tau}{\Gamma \vdash_s S \circ \texttt{swap}\ v,\ [\text{–}]\ \texttt{in}\ e'' \;:\; \texttt{name} \multimap \tau'}$$

$$\text{swap-3}\frac{\Gamma \vdash_s S \;:\; \tau \multimap \tau' \quad \Gamma \vdash v \;:\; \texttt{name} \quad \Gamma \vdash v' \;:\; \texttt{name}}{\Gamma \vdash_s S \circ \texttt{swap}\ v,\ v'\ \texttt{in}\ [\text{–}] \;:\; \tau \multimap \tau'}$$

$$\text{app-l}\frac{\Gamma \vdash_s S \;:\; \tau' \multimap \tau'' \quad \Gamma \vdash e \;:\; \tau}{\Gamma \vdash_s S \circ [\text{–}]\ e \;:\; (\tau \to \tau') \multimap \tau''} \qquad \text{app-r}\frac{\Gamma \vdash_s \;:\tau' \multimap \tau'' \quad \Gamma \vdash v \;:\; \tau \to \tau'}{\Gamma \vdash_s S \circ v\ [\text{–}] \;:\; \tau \multimap \tau''}$$

$$\text{if-l}\frac{\Gamma \vdash_s S \;:\; \tau' \multimap \tau \quad \Gamma \vdash e \;:\; \texttt{name} \quad \Gamma \vdash e' \;:\; \tau' \quad \Gamma \vdash e'' \;:\; \tau'}{\Gamma \vdash_s S \circ \texttt{if}\ [\text{–}]\ \texttt{=}\ e\ \texttt{then}\ e'\ \texttt{else}\ e'' \;:\; \texttt{name} \multimap \tau}$$

$$\text{if-r}\frac{\Gamma \vdash_s S \;:\; \tau' \multimap \tau \quad \Gamma \vdash v \;:\; \texttt{name} \quad \Gamma \vdash e' \;:\; \tau' \quad \Gamma \vdash e'' \;:\; \tau'}{\Gamma \vdash_s S \circ \texttt{if}\ v\ \texttt{=}\ [\text{–}]\ \texttt{then}\ e'\ \texttt{else}\ e'' \;:\; \texttt{name} \multimap \tau}$$

$$\text{let}\frac{\Gamma \vdash_s S \;:\; \tau' \multimap \tau'' \quad \Gamma, x \;:\; \tau \vdash e \;:\; \tau'}{\Gamma \vdash_s S \circ \texttt{let}\ x\ \texttt{=}\ [\text{–}]\ \texttt{in}\ e \;:\; \tau \multimap \tau''}$$

$$\text{let-pair}\frac{\Gamma \vdash_s S \;:\; \tau \multimap \tau' \quad \Gamma, x \;:\; \tau_1, x' \;:\; \tau_2 \vdash e \;:\; \tau}{\Gamma \vdash_s S \circ \texttt{let}\ (x,\ x')\ \texttt{=}\ [\text{–}]\ \texttt{in}\ e \;:\; \tau_1 \times \tau_2 \multimap \tau'}$$

$$\text{let-abst}\frac{\Gamma \vdash_s \;:\tau' \multimap \tau'' \quad \Gamma, x \;:\; \texttt{name}, x' \;:\; \tau \vdash e \;:\; \tau'}{\Gamma \vdash_s S \circ \texttt{let <<}x\texttt{>>}x'\ \texttt{=}\ [\text{–}]\ \texttt{in}\ e \;:\; \texttt{<<name>>}\tau \multimap \tau''}$$

$$\text{match}\frac{\Gamma \vdash_s S \;:\; \tau \multimap \tau' \quad \text{for all } 1 \leq i \leq K, \Gamma, x_i \;:\; \sigma_i \vdash e_i \;:\; \tau}{\Gamma \vdash_s S \circ \texttt{match}\ [\text{–}]\ \texttt{with}\ \cdots \texttt{|}\ \mathtt{C}_k(x_k)\ \texttt{->}\ e_k\ \texttt{|}\ \cdots \;:\; \delta \multimap \tau'}$$

Figure 3.6: Typing rules for frame stacks.

the finitely many atoms previously generated. When we need to generate a new atom, we always know that there will be cofinitely many left; and furthermore, by the next lemma then we know that *any* such one will do. This is an example of the 'some/any' duality expressed by use of the И quantifier in FM-set theory[20, 18].

Lemma 3.4.2 (Equivariance for the termination relation). *For atoms a, a' then $\langle S, e\rangle\downarrow$ implies $\langle (a\ a')\cdot S, (a\ a')\cdot e\rangle\downarrow$, where $(a\ a')\cdot S$ is that frame stack obtained by recursively swapping all occurrences of a and a' in S.* □

Lemma 3.4.3 (Splitting frame stacks). *If $\langle S \mathbin{@} S', e\rangle\downarrow$ then $\langle S', e\rangle\downarrow$. Therefore in particular, $\langle S, e\rangle\downarrow$ implies that $\langle [], e\rangle\downarrow$.* □

As an example of Lemma 3.4.2, suppose that we wish to determine whether the `fresh` expression terminates in some frame stack S of type $\texttt{name} \multimap \_$. We see from the definition of the termination relation that this holds just when $\langle S, a\rangle\downarrow$ does, for some a not occurring in the atoms of S. Therefore by Lemma 3.4.2 it follows that for any other such a', $\langle (a\ a')\cdot S, (a\ a')\cdot a\rangle\downarrow$ holds; so $\langle S, a'\rangle\downarrow$ does also.

3.4.2 Relationship to the big-step semantics

We now wish to establish a link between the frame stack semantics and the big-step semantics.

Lemma 3.4.4. *Let e be a closed expression of type τ. Then $\overline{a}, e \Downarrow v, \overline{a}'$ implies that for all frame stacks S of type $\tau \multimap \_$, $\mathrm{atms}(S) \subseteq \overline{a} \wedge \langle S, v\rangle\downarrow$ implies that $\langle S, e\rangle\downarrow$.*

Proof. By showing that the property

$$\Phi(\overline{a}, e, v, \overline{a}') \stackrel{\text{def}}{=} \forall S : \tau \multimap \_ .\ \vdash e : \tau \wedge \mathrm{atms}(S) \subseteq \overline{a} \wedge \langle S, v\rangle\downarrow \Rightarrow \langle S, e\rangle\downarrow$$

is closed under the axioms and rules inductively defining the evaluation relation $\Downarrow$. Note that Lemma 3.3.1 implies that we can always take $\overline{a}' \supseteq \overline{a}$ and $\mathrm{atms}(v) \subseteq \overline{a}'$; moreover, we always have that $\mathrm{atms}(e) \subseteq \overline{a}$ since $\Downarrow$ is only defined in the cases when this holds.

Let us now give the two interesting cases: those for fresh name creation and deconstruction of abstraction values. The remaining ones are routine.

▶ *Case (fresh).* For some finite set of atoms $\overline{a}$ and some $a \in \mathbb{A} \setminus \overline{a}$ we want $\Phi(\overline{a}, \texttt{fresh}, a, \overline{a} \uplus \{a\})$. The assumption $\mathrm{atms}(S) \subseteq \overline{a}$ implies that $a \notin \mathrm{atms}(S)$; combining this with the other assumption that $\langle S, a\rangle\downarrow$ gives us $\langle S, \texttt{fresh}\rangle\downarrow$.

▶ *Case (let-abst).* We wish to show that $\Phi(\overline{a}, \texttt{let <<}x\texttt{>>}x' \texttt{ = } e \texttt{ in } e', v, \overline{a}'')$ holds, for finite sets of atoms $\overline{a} \subseteq \overline{a}' \subseteq \overline{a}''$. As induction hypotheses we have that $\Phi(\overline{a}, e, \texttt{<<}a\texttt{>>}v', \overline{a}')$ and $\Phi(\overline{a}', e'[a'/x, (a\ a')\cdot v'/x'], v, \overline{a}'')$ for $a \in \mathbb{A}$ and any $a' \in \mathbb{A} \setminus \overline{a}'$. Expanding these we obtain as assumptions that

$$\forall S_1.\ \mathrm{atms}(S_1) \subseteq \overline{a} \wedge \langle S_1, \texttt{<<}a\texttt{>>}v'\rangle\downarrow \Rightarrow \langle S_1, e\rangle\downarrow; \text{ and} \tag{3.1}$$
$$\forall S_2.\ \mathrm{atms}(S_2) \subseteq \overline{a}' \wedge \langle S_2, v\rangle\downarrow \Rightarrow \langle S_2, e'[a'/x, (a\ a')\cdot v'/x']\rangle\downarrow. \tag{3.2}$$

For S with $\mathrm{atms}(S) \subseteq \overline{a}$ we wish to show that $\langle S, \texttt{let <<}x\texttt{>>}x' \texttt{ = } e \texttt{ in } e'\rangle\downarrow$ under the assumption that $\langle S, v\rangle\downarrow$. It therefore suffices to show that $\langle S \circ \texttt{let <<}x\texttt{>>}x' \texttt{ = [–] in } e', e\rangle\downarrow$ holds. This follows from (3.1) if we can show

$$\mathrm{atms}(S \circ \texttt{let <<}x\texttt{>>}x' \texttt{ = [–] in } e') \subseteq \overline{a}; \text{ and} \tag{3.3}$$
$$\langle S \circ \texttt{let <<}x\texttt{>>}x' \texttt{ = [–] in } e', \texttt{<<}a\texttt{>>}v'\rangle\downarrow. \tag{3.4}$$

We have $\mathrm{atms}(\texttt{let <<}x\texttt{>>}x' \texttt{ = } e \texttt{ in } e') \subseteq \overline{a}$ (from the earlier observation before the first proof case), which shows (3.3) since $\mathrm{atms}(S) \subseteq \overline{a}$ holds by assumption. To see (3.4) it suffices to show that

$$\langle S, e'[a''/x, (a\ a'')\cdot v'/x']\rangle\downarrow \tag{3.5}$$

holds, where $a'' \in \mathbb{A} \setminus \mathrm{atms}(S, a, v', e')$. Since $\overline{a}' \supseteq \overline{a}$ then $\mathrm{atms}(S) \subseteq \overline{a}'$. But $\mathrm{atms}(\texttt{<<}a\texttt{>>}v') \subseteq \overline{a}'$ (again from the earlier observations) and therefore $\mathrm{atms}(a, v') \subseteq \overline{a}'$. However, since we have that $\mathrm{atms}(\texttt{let <<}x\texttt{>>}x' \texttt{ = } e \texttt{ in } e') \subseteq \overline{a}$ then $\mathrm{atms}(e') \subseteq \overline{a}'$. Therefore it is certainly the case that $a'' \in \mathbb{A}\setminus\overline{a}'$ implies $a'' \in \mathbb{A}\setminus\mathrm{atms}(S, a, v', e')$. We may therefore choose any $a'' \in \mathbb{A}\setminus\overline{a}'$ and attempt to apply (3.2) (using this a'' as the a' in that hypothesis) to conclude (3.5). It therefore remains to show that firstly, $\mathrm{atms}(S) \subseteq \overline{a}'$ and secondly, $\langle S, v\rangle\downarrow$. However the first of these is satisfied since $\overline{a}' \supseteq \overline{a}$ and the second of these follows by earlier assumption. □

Lemma 3.4.5. *Let e be a closed expression of type τ and let S be a closed frame stack of type $\tau \multimap \_$. Then $\langle S, e\rangle\downarrow$ implies that for all $\overline{a} \supseteq \mathrm{atms}(e)$ there exists a value v and some $\overline{a}' \supseteq \overline{a}$ such that $\overline{a}, e \Downarrow v, \overline{a}' \wedge \langle S, v\rangle\downarrow$.*

Proof. By showing that the property

$$\begin{aligned}\Phi(S, e) \stackrel{\text{def}}{=} \vdash e : \tau \wedge \vdash_s S : \tau \multimap \_ \wedge \langle S, e\rangle\downarrow \Rightarrow \\ \forall \overline{a}.\ \overline{a} \supseteq \mathrm{atms}(e) \Rightarrow \exists v, \overline{a}'.\ \overline{a}' \supseteq \overline{a} \wedge \overline{a}, e \Downarrow v, \overline{a}' \wedge \langle S, v\rangle\downarrow\end{aligned}$$

$$\text{empty}\frac{}{\langle [], v\rangle\downarrow} \qquad \text{con}\frac{\langle S, \mathtt{C}_k\,(v)\rangle\downarrow}{\langle S \circ \mathtt{C}_k\,([-]), v\rangle\downarrow} \qquad \text{pair-l}\frac{\langle S \circ \mathtt{(}v\mathtt{,}\ [-]\mathtt{)}, e\rangle\downarrow}{\langle S \circ \mathtt{([-],}\ e\mathtt{)}, v\rangle\downarrow} \qquad \text{pair-r}\frac{\langle S, \mathtt{(}v'\mathtt{,}\ v\mathtt{)}\rangle\downarrow}{\langle S \circ \mathtt{(}v'\mathtt{,}\ [-]\mathtt{)}, v\rangle\downarrow}$$

$$\text{abst-l}\frac{\langle S \circ \mathtt{<<}v\mathtt{>>}[-], e\rangle\downarrow}{\langle S \circ \mathtt{<<[-]>>}e, v\rangle\downarrow} \qquad \text{abst-r}\frac{\langle S, \mathtt{<<}v\mathtt{>>}v'\rangle\downarrow}{\langle S \circ \mathtt{<<}v\mathtt{>>[-]}, v'\rangle\downarrow}$$

$$\text{swap-1}\frac{\langle S \circ \mathtt{swap}\ a\mathtt{,}\ [-]\ \mathtt{in}\ e'', e'\rangle\downarrow}{\langle S \circ \mathtt{swap}\ [-]\mathtt{,}\ e'\ \mathtt{in}\ e'', a\rangle\downarrow}$$

$$\text{swap-2}\frac{\langle S \circ \mathtt{swap}\ a\mathtt{,}\ a'\ \mathtt{in}\ [-], e''\rangle\downarrow}{\langle S \circ \mathtt{swap}\ a\mathtt{,}\ [-]\ \mathtt{in}\ e'', a'\rangle\downarrow} \qquad \text{swap-3}\frac{\langle S, (a\ a')\cdot v\rangle\downarrow}{\langle S \circ \mathtt{swap}\ a\mathtt{,}\ a'\ \mathtt{in}\ [-], v\rangle\downarrow}$$

$$\text{app-l}\frac{\langle S \circ v\ [-], e\rangle\downarrow}{\langle S \circ [-]\ e, v\rangle\downarrow} \qquad \text{app-r}\frac{\langle S, e[v/f, v'/x]\rangle\downarrow}{\langle S \circ v\ [-], v'\rangle\downarrow}\ \text{if } v = \mathtt{fun}\ f(x)\ \mathtt{=}\ e$$

$$\text{let}\frac{\langle S, e[v/x]\rangle\downarrow}{\langle S \circ \mathtt{let}\ x\ \mathtt{=}\ [-]\ \mathtt{in}\ e, v\rangle\downarrow} \qquad \text{let-pair}\frac{\langle S, e[v/x, v'/x']\rangle\downarrow}{\langle S \circ \mathtt{let\ (}x\mathtt{,}\ x'\mathtt{)\ =}\ [-]\ \mathtt{in}\ e, \mathtt{(}v\mathtt{,}\ v'\mathtt{)}\rangle\downarrow}$$

$$\text{let-abst}\frac{\langle S, e[a'/x, ((a\ a')\cdot v)/x']\rangle\downarrow}{\langle S \circ \mathtt{let\ <<}x\mathtt{>>}x'\ \mathtt{=}\ [-]\ \mathtt{in}\ e, \mathtt{<<}a\mathtt{>>}v\rangle\downarrow}\ a' \in \mathbb{A} \setminus \mathrm{atms}(S, a, v, e)$$

$$\text{if-l}\frac{\langle S \circ \mathtt{if}\ a\ \mathtt{=}\ [-]\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2, e'\rangle\downarrow}{\langle S \circ \mathtt{if}\ [-]\ \mathtt{=}\ e'\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2, a\rangle\downarrow}$$

$$\text{if-r-1}\frac{\langle S, e_1\rangle\downarrow}{\langle S \circ \mathtt{if}\ a\ \mathtt{=}\ [-]\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2, a'\rangle\downarrow}\ \text{if } a = a'$$

$$\text{if-r-2}\frac{\langle S, e_2\rangle\downarrow}{\langle S \circ \mathtt{if}\ a\ \mathtt{=}\ [-]\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2, a'\rangle\downarrow}\ \text{if } a \neq a'$$

$$\text{match}\frac{v = \mathtt{C}_k\,(v_k)\ \wedge\ \langle S, e_k[v_k/x_k]\rangle\downarrow}{\langle S \circ \mathtt{match}\ [-]\ \mathtt{with}\ \mathtt{C}_1\,(x_1)\ \mathtt{->}\ e_1\mathtt{|}\ \cdots\mathtt{|}\ \mathtt{C}_K\,(x_K)\ \mathtt{->}\ e_K\ , v\rangle\downarrow}$$

$$\text{nc-con}\frac{\langle S \circ \mathtt{C}_k\,([-]), e\rangle\downarrow}{\langle S, \mathtt{C}_k\,(e)\rangle\downarrow} \qquad \text{nc-fresh}\frac{\langle S, a\rangle\downarrow}{\langle S, \mathtt{fresh}\rangle\downarrow}\ a \in \mathbb{A} \setminus \mathrm{atms}(S) \qquad \text{nc-pair}\frac{\langle S \circ \mathtt{([-],}\ e'\mathtt{)}, e\rangle\downarrow}{\langle S, \mathtt{(}e\mathtt{,}\ e'\mathtt{)}\rangle\downarrow}$$

$$\text{nc-abst}\frac{\langle S \circ \mathtt{<<[-]>>}e', e\rangle\downarrow}{\langle S, \mathtt{<<}e\mathtt{>>}e'\rangle\downarrow} \qquad \text{nc-swap}\frac{\langle S \circ \mathtt{swap}\ [-]\mathtt{,}\ e'\ \mathtt{in}\ e'', e\rangle\downarrow}{\langle S, \mathtt{swap}\ e\mathtt{,}\ e'\ \mathtt{in}\ e''\rangle\downarrow}$$

$$\text{nc-app}\frac{\langle S \circ [-]\ e', e\rangle\downarrow}{\langle S, e\ e'\rangle\downarrow} \qquad \text{nc-let}\frac{\langle S \circ \mathtt{let}\ x\ \mathtt{=}\ [-]\ \mathtt{in}\ e', e\rangle\downarrow}{\langle S, \mathtt{let}\ x\ \mathtt{=}\ e\ \mathtt{in}\ e'\rangle\downarrow}$$

$$\text{nc-let-pair}\frac{\langle S \circ \mathtt{let\ (}x\mathtt{,}\ x'\mathtt{)\ =}\ [-]\ \mathtt{in}\ e', e\rangle\downarrow}{\langle S, \mathtt{let\ (}x\mathtt{,}\ x'\mathtt{)\ =}\ e\ \mathtt{in}\ e'\rangle\downarrow}$$

$$\text{nc-let-abst}\frac{\langle S \circ \mathtt{let\ <<}x\mathtt{>>}x'\ \mathtt{=}\ [-]\ \mathtt{in}\ e', e\rangle\downarrow}{\langle S, \mathtt{let\ <<}x\mathtt{>>}x'\ \mathtt{=}\ e\ \mathtt{in}\ e'\rangle\downarrow}$$

$$\text{nc-if}\frac{\langle S \circ \mathtt{if}\ [-]\ \mathtt{=}\ e'\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2, e\rangle\downarrow}{\langle S, \mathtt{if}\ e\ \mathtt{=}\ e'\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\rangle\downarrow}$$

$$\text{nc-match}\frac{\langle S \circ \mathtt{match}\ [-]\ \mathtt{with}\ \mathtt{C}_1\,(x_1)\ \mathtt{->}\ e_1\mathtt{|}\ \cdots\mathtt{|}\ \mathtt{C}_K\,(x_K)\ \mathtt{->}\ e_K\ , e\rangle\downarrow}{\langle S, \mathtt{match}\ e\ \mathtt{with}\ \mathtt{C}_1\,(x_1)\ \mathtt{->}\ e_1\mathtt{|}\ \cdots\mathtt{|}\ \mathtt{C}_K\,(x_K)\ \mathtt{->}\ e_K\ \rangle\downarrow}$$

Figure 3.7: Termination relation.

is closed under the axioms and rules inductively defining the termination relation. The proof splits into two parts, depending on whether or not e is a canonical form. It is easily seen that in the case where e is indeed canonical, then $\Phi(S, e)$ holds trivially. When e is not canonical, we must examine individual cases. Here we give the same two cases as we did for Lemma 3.4.4; the others are again routine. The two cases we pick highlight how the equivariance property of the termination relation (viz. Lemma 3.4.2) must be exploited during calculations.

▶ *Case (nc-fresh).* Assume $\langle S, \texttt{fresh}\rangle\downarrow$ and take any finite set of atoms $\overline{a}$. Then $\overline{a} \supseteq \mathrm{atms}(\texttt{fresh})$ and for any $a \notin \mathrm{atms}(S)$, $\langle S, a\rangle\downarrow$ holds. Now take any $a' \in \mathbb{A} \setminus \mathrm{atms}(S) \setminus \overline{a} \setminus \{a\}$. Then $(a\ a') \cdot S = S$ and by Lemma 3.4.2, $\langle S, a'\rangle\downarrow$ holds. Since $a \notin \overline{a}$ it also follows that $\overline{a}, \texttt{fresh} \Downarrow a', \overline{a} \uplus \{a'\}$ as required.

▶ *Case (nc-let-abst).* Make the assumptions that $\langle S, \texttt{let <<}x\texttt{>>}x' \texttt{ = } e \texttt{ in } e'\rangle\downarrow$ and $\overline{a} \supseteq \mathrm{atms}(\texttt{let <<}x\texttt{>>}x' \texttt{ = } e \texttt{ in } e')$. The termination judgement must have been derived by knowing $\langle S \circ \texttt{let <<}x\texttt{>>}x' \texttt{ = } [\text{–}] \texttt{ in } e', e\rangle\downarrow$; since $\overline{a} \supseteq \mathrm{atms}(e)$ it follows (by the first induction hypothesis) that there exists a value v'' and some $\overline{a}' \supseteq \overline{a}$ with $\overline{a}, e \Downarrow v'', \overline{a}'$ and a termination judgement $\langle S \circ \texttt{let <<}x\texttt{>>}x' \texttt{ = } [\text{–}] \texttt{ in } e', v''\rangle\downarrow$; Lemma 3.3.4 tells us that we can write v'' as $\texttt{<<}a\texttt{>>}v'$. This termination judgement must in turn have been derived by knowing that $\langle S, e'[a'/x, (a\ a') \cdot v']\rangle\downarrow$, for some $a' \in \mathbb{A} \setminus \mathrm{atms}(S, e', a, v')$. Taking any $a'' \in \mathbb{A} \setminus \mathrm{atms}(S, e', a, a', v') \setminus \overline{a}'$, Lemma 3.4.2 tells us that $\langle S, e'[a''/x, (a\ a'') \cdot v']\rangle\downarrow$. It follows (by the second induction hypothesis) that there exists a v and some $\overline{a}'' \supseteq \overline{a}'$ with $\overline{a}', e'[a''/x, (a\ a'') \cdot v'] \Downarrow v, \overline{a}''$ and $\langle S, v\rangle\downarrow$. It follows that the evaluation judgement $\overline{a}, \texttt{let <<}x\texttt{>>}x' \texttt{ = } e \texttt{ in } e' \Downarrow v, \overline{a}''$ holds since $a'' \notin \overline{a}'$. □

The following theorem, with associated corollary, now follows immediately by combining Lemmata 3.4.4 and 3.4.5.

Theorem 3.4.6 (Equivalence of dynamic semantics). *Let S be a frame stack of type $\tau \multimap \_$ and e be a closed expression of type τ. Then $\langle S, e\rangle\downarrow$ holds iff there exists $\overline{a} \supseteq \mathrm{atms}(S, e)$ with $\overline{a}, e \Downarrow v, \overline{a}'$ and $\langle S, v\rangle\downarrow$, for some value v and a finite set of atoms $\overline{a}' \supseteq \overline{a}$.* □

Corollary 3.4.7. *Let e be a closed expression of type τ. Then $\langle [], e\rangle\downarrow$ holds just when there exists $\overline{a}' \supseteq \overline{a} \supseteq \mathrm{atms}(e)$ and a value v with $\overline{a}, e \Downarrow v, \overline{a}'$.* □

3.5 Environment style semantics

In this section, we very briefly review another variety of semantics: that given in *environment style*. Such a semantics involves the propagation of what we shall call a *value environment*: a finite map from value identifiers to values. The space of values may be disjoint from the space of expressions and the environments are used to look up values of previously-introduced identifiers.

As an example, the evaluation rule

$$\frac{\overline{a}, e \Downarrow v', \overline{a}' \qquad \overline{a}', e'[v'/x] \Downarrow v, \overline{a}''}{\overline{a}, \texttt{let } x \texttt{ = } e \texttt{ in } e' \Downarrow v, \overline{a}''}$$

from §3.3 could be written in environment style as follows:

$$\frac{\overline{a}, E \vdash e \Downarrow v', \overline{a}' \qquad \overline{a}', E[x \mapsto v'] \vdash e' \Downarrow v, \overline{a}''}{\overline{a}, E \vdash \texttt{let } x \texttt{ = } e \texttt{ in } e' \Downarrow v, \overline{a}''}$$

where E stands for a value environment and $E[x \mapsto v']$ is that value environment mapping x to v' and otherwise acting as E.

It should be evident that an environment style semantics corresponds more closely to how an implementation might work. A classic example of such a semantics is that given by Milner et al.[36] in the definition of the Standard ML language.

For the most part, this thesis uses the familiar 'substituted-in' style of semantics which we have seen earlier in this chapter. (In that setting, the canonical forms must be a subset of the expressions of the language.) There is no particular reason for choosing one way or the other for the forthcoming correctness results. However, in Chapter 7 we make use of an environment style semantics in a setting where it greatly simplifies the presentation.

C	::=	$[\text{–}]$	hole
		x	value identifier
		`()`	unit
		a	atom
		$\mathtt{C}_k(C)$	data construction
		(C, C)	pairing
		`fresh`	fresh name creation
		$\mathtt{<<}C\mathtt{>>}C$	abstraction
		`swap` C`,` C `in` C	atom-swapping
		`fun` $x(x)$ `=` C	recursive function
		$C\ C$	function application
		`if` C `=` C `then` C `else` C	conditional
		`let` x `=` C `in` C	value binding
		`let` (x, x) `=` C `in` C	pair deconstruction
		`let` $\mathtt{<<}x\mathtt{>>}x$ `=` C `in` C	abstraction deconstruction
		`match` C `with`	data value deconstruction
		$\mathtt{C}_1(x_1)$ `->` C_1	
		`|` $\cdots$	
		`|` $\mathtt{C}_K(x_K)$ `->` C_K	

Figure 3.8: Contexts.

3.6 Notions of observational equivalence

Soon we will be talking about the *observational equivalence* of Mini-FreshML expressions: when does one expression behave the same as another, in some sense? At this stage it is appropriate to pick a particular notion of equivalence with which to work. We are actually going to use what is regarded as something of a benchmark for observational equivalence: the notion of *contextual equivalence*[3]. Two expressions are said to be contextually equivalent if they may be freely exchanged within the source text of some enclosing program without affecting the program's observable results.

Definition 3.6.1 (Contexts). A context C consists of an expression containing one or more 'holes', as given in Figure 3.8. For contexts C and C', write $C[C']$ for the context formed by inserting C' into the holes in C. For an expression e, write $C[e]$ for the expression formed by inserting e into the holes in C. Note that both of these operations[4] may involve binders in C capturing identifiers in C' (resp. e); thus, *contexts are not identified up to α-conversion*. $\diamond$

Definition 3.6.2. For frame stacks S and frames $\mathcal{F}$ then define $\mathcal{C}txt(\text{–})$ by induction on the size of S as follows.

$$\mathcal{C}txt([]) \stackrel{\text{def}}{=} [\text{–}] \qquad \mathcal{C}txt(S \circ \mathcal{F}) \stackrel{\text{def}}{=} (\mathcal{C}txt(S))[\mathcal{F}]. \quad \diamond$$

Lemma 3.6.3. *For a closed frame stack S of type $\tau \multimap \_$ and a closed expression e of type τ,*

$$\langle S, e\rangle\downarrow \Leftrightarrow \langle [], (\mathcal{C}txt(S))[e]\rangle\downarrow. \tag{3.6}$$

Proof. By a monotonous induction on the structure of S, whose details we omit. $\square$

Definition 3.6.4 (Contextual preorder, contextual equivalence). The 4-tuple (Γ, e, e', τ) is said to be in the type-respecting relation of *contextual preorder* iff e and e' may be assigned type τ in typing context Γ and for all contexts C such that $C[e]$ and $C[e']$ are typeable closed expressions,

$$(\forall \overline{a} \supseteq \mathrm{atms}(C[e]).\ \exists \overline{a}' \supseteq \overline{a}.\ \exists v \in \mathrm{Val}_\tau.\ \overline{a}, C[e] \Downarrow v, \overline{a}') \Rightarrow$$

[3] Specified in a traditional way, no less: for an arguably more modern exposition see [44], for example.

[4] Such operations are a good example of metaprogramming tasks which are significantly simplified by the presence of freshness features in the metalanguage.

$$(\forall \overline{a} \supseteq \mathrm{atms}(C[e']).\ \exists \overline{a}' \supseteq \overline{a}.\ \exists v' \in \mathrm{Val}_\tau.\ \overline{a}, C[e'] \Downarrow v', \overline{a}').$$

Write $\Gamma \vdash e \prec_{\mathrm{ctx}} e' : \tau$ iff (Γ, e, e', τ) is in the relation. The relation of *contextual equivalence,* denoted by $\Gamma \vdash e \approx_{\mathrm{ctx}} e' : \tau$, is the symmetrisation of contextual preorder. Let us just write $e \approx_{\mathrm{ctx}} e'$ iff e and e' are closed contextually-equivalent expressions. $\diamond$

It is an immediate consequence of Theorem 3.4.6 that we can characterise contextual equivalence using frame stack semantics in the following way.

Lemma 3.6.5 ($\approx_{\mathrm{ctx}}$ in terms of termination). *The contextual pre-order $\Gamma \vdash e \prec_{\mathrm{ctx}} e' : \tau$ holds just when for all contexts C such that $C[e]$ and $C[e']$ are typeable closed expressions, $\langle [], C[e]\rangle\!\downarrow$ implies that $\langle [], C[e']\rangle\!\downarrow$. Therefore $\Gamma \vdash e \approx_{\mathrm{ctx}} e' : \tau$ just when for all such contexts C, $\langle [], C[e]\rangle\!\downarrow \Leftrightarrow \langle [], C[e']\rangle\!\downarrow$.* $\square$

Now whilst contextual equivalence is a powerful notion which is convenient for establishing further results, it can be difficult to work with due to the necessity to quantify over all possible contexts. We manage to get around such difficulties by calculating with another notion of equivalence: Mason and Talcott's notion of 'CIU-equivalence'[29]. In §5.5 we will prove that this coincides with Mini-FreshML contextual equivalence.

Definition 3.6.6 (CIU-preorder, CIU-equivalence). The 4-tuple (Γ, e, e', τ) is said to be in the type-respecting relation of *CIU-preorder* iff $\Gamma \vdash e : \tau$, $\Gamma \vdash e' : \tau$ and for all substitutions ψ with $\vdash \psi : \Gamma$ and all closed frame stacks S of type $\tau \multimap \_$ then $\langle S, e[\psi]\rangle\!\downarrow \Rightarrow \langle S, e'[\psi]\rangle\!\downarrow$. Write $\Gamma \vdash e \prec_{\mathrm{ciu}} e' : \tau$ iff (Γ, e, e', τ) is in the relation. The symmetrisation is called *CIU-equivalence,* written $\Gamma \vdash e \approx_{\mathrm{ciu}} e' : \tau$. Write $e \approx_{\mathrm{ciu}} e'$ iff e and e' are closed CIU-equivalent expressions. $\diamond$

3.7 Correctness for Mini-FreshML

In this section we shall examine some desirable Mini-FreshML correctness properties which will be proved later in Chapter 5. These properties centre around the behaviour of Mini-FreshML expressions which are being used to represent, or compute, values of some object-language syntax.

For simplicity, we will consider an object language whose terms $t \in \Lambda$ are those of the untyped λ-calculus,

$$t ::= x \mid \lambda x.\ t \mid t\ t$$

where x ranges over the countably infinite set of variables VId. Whilst this example is simple, it captures the essential features of object languages specified by a certain notion of *binding signature*[17, 18] to which our theory can be generalised. Providing the operations of the binding signature do not involve function space constructions then the syntax of such a language may be specified by an algebraic datatype[5]. In this particular case, the suitable datatype declaration is:

$$\texttt{type}\ \delta\ \texttt{=}\ \mathtt{C}_1\ \texttt{of name |}\ \mathtt{C}_2\ \texttt{of <<name>>}\delta\ \texttt{|}\ \mathtt{C}_3\ \texttt{of}\ \delta \times \delta.$$

Let us write `Var` for $\mathtt{C}_1$, `Lam` for $\mathtt{C}_2$, `App` for $\mathtt{C}_3$ and `lam` for δ. Furthermore, write $\mathrm{vars}(t)$ for the finite set of all variables occurring within a term t. Extend this notation to handle multiple terms in the obvious manner. Given value identifiers x, $x' \in \mathrm{VId}$ and a λ-term t then write $(x\ x') \cdot t$ for that term obtained by swapping *all* occurrences of x and x' throughout t.

We shall be particularly concerned with the relationships which exist between Mini-FreshML expressions and α-equivalence classes of object-level terms. We use the characterisation of α-equivalence given by Gabbay and Pitts[20], which is defined by structural recursion on λ-terms as follows.

Definition 3.7.1 (α-equivalence).

$$\frac{}{x \equiv_\alpha x}\ x \in \mathrm{VId} \tag{3.7}$$

$$\frac{(x\ x'') \cdot t \equiv_\alpha (x'\ x'') \cdot t'}{\lambda x.\ t \equiv_\alpha \lambda x'.\ t'}\ x'' \in \mathrm{VId} \setminus \mathrm{vars}(x, x', t, t') \tag{3.8}$$

[5] That is, one not involving the function space constructor: note that this notion of 'algebraic' differs from the usual one in the sense that we allow `<<name>>`τ types to appear.

$$\frac{t_1 \equiv_\alpha t_1' \qquad t_2 \equiv_\alpha t_2'}{t_1\ t_2 \equiv_\alpha t_1'\ t_2'}. \quad \diamond \tag{3.9}$$

Note that α-equivalence is closed under swapping: if $t \equiv_\alpha t'$ then $(x\ x') \cdot t \equiv_\alpha (x\ x') \cdot t'$ for *any* value identifiers x, $x' \in \mathrm{VId}$. Such identities do not hold when one uses arbitrary *renamings* as opposed to swappings. For example the open terms $\lambda x.\ x\ y$ and $\lambda z.\ z\ y$ are α-equivalent, but given a free identifier x we obtain that $(\lambda x.\ x\ y)[y \mapsto x] = \lambda x.\ x\ x \not\equiv_\alpha \lambda z.\ z\ x = (\lambda z.\ z\ y)[y \mapsto x]$ (writing $[y \mapsto x]$ for the *non*-capture-avoiding substitution of x for y). However, $(x\ y) \cdot (\lambda x.\ x\ y) = \lambda y.\ y\ x \equiv_\alpha \lambda z.\ z\ x = (x\ y) \cdot (\lambda z.\ z\ y)$. Good properties like this did indeed inspire development of the original permutation model of name binding[20].

Given some λ-term t, we may encode it as a Mini-FreshML expression by using the following translation function.

Definition 3.7.2 (Translation to expressions). For each λ-term t, define a Mini-FreshML expression $[t]_{\mathrm{e}} \in \mathrm{Exp}_\delta$ by induction on the structure of t as follows.

$$\begin{aligned} [x]_{\mathrm{e}} &\stackrel{\mathrm{def}}{=} \texttt{Var(}x\texttt{)} \\ [\lambda x.\ t]_{\mathrm{e}} &\stackrel{\mathrm{def}}{=} \texttt{let } x \texttt{ = fresh in Lam(<<}x\texttt{>>)}[t]_{\mathrm{e}}\texttt{)} \\ [t\ t']_{\mathrm{e}} &\stackrel{\mathrm{def}}{=} \texttt{App(}[t]_{\mathrm{e}}\texttt{, }[t']_{\mathrm{e}}\texttt{)} \end{aligned}$$

Recall we identify expressions up to α-equivalence of bound value identifiers. If t is a closed (resp. open) term, then $[t]_{\mathrm{e}}$ is a closed (resp. open) value. $\diamond$

Lemma 3.7.3. $[-]_{\mathrm{e}}$ *is equivariant: if* $x, x' \in \mathrm{VId}$ *then* $(x\ x') \cdot [t]_{\mathrm{e}} = [(x\ x') \cdot t]_{\mathrm{e}}$. $\square$

Using our denotational semantics we will be able to prove the following facts about Mini-FreshML. The first of these is the most important result of all, for it shows how contextual equivalence at the meta-level coincides with α-equivalence at the object level.

Fact 3.7.4 (Correctness for expressions). *For λ-terms t, t' with free variables contained in a finite set $\{x_0, \cdots, x_n\} \subseteq \mathrm{VId}$,*

$$t \equiv_\alpha t' \Leftrightarrow \{x_0 \texttt{ : name}, \cdots, x_n \texttt{ : name}\} \vdash [t]_{\mathrm{e}} \approx_{\mathrm{ctx}} [t']_{\mathrm{e}} \texttt{ : lam}. \quad \diamond$$

Fact 3.7.5 (Form of expressions). *For a closed Mini-FreshML expression e of type* `lam` *and a divergent term Ω, then either $e \approx_{\mathrm{ctx}} \Omega$ or there exists a closed value v of type* `lam` *with*

$$e \approx_{\mathrm{ctx}} \texttt{let } x_1 \texttt{ = fresh in } \cdots \texttt{ let } x_n \texttt{ = fresh in } v,$$

for value identifiers $\{x_1, \ldots, x_n\}$.

These results are neither quick nor straightforward to prove. In order to show their correctness, we shall develop a new variety of domain theory and then exploit it to obtain a denotational semantics for Mini-FreshML. By relating this to the operational semantics, we will be able to derive the desired results at the end of Chapter 5.

4 A domain theory for names

'There is an infinite set A that is not too big.' —von Neumann

IN THIS CHAPTER, we develop a variant of classical domain theory[3, 52] which is good for reasoning about names and name binding. This theory forms the basis of our denotational model of Mini-FreshML in Chapter 5. The domain theory centres around the permutation model of name binding developed by Pitts and Gabbay[20, 18], although there is an important difference between their presentation and that given here. For those authors take as their foundation an axiomatisation of *FM-set theory* and develop constructions fully internalised within that world. In this thesis, we take a somewhat less radical approach and make constructions in standard ZF-set theory which parallel those in FM-set theory. This enables us to exploit the novel features offered by FM whilst still retaining a familiar logical foundation.

FM-domain theory provides a relatively uncomplicated setting for reasoning about names and name binding. In particular, all dependencies on parameterising name sets may be made implicit—removing any need to index over 'possible worlds' as seen so widely in approaches based on functor category semantics[17, 66]. The various constructions which we make in FM-domain theory are generally quite straightforward (although occasionally rather subtle) and benefit from not being complicated by such indexing. Although as we shall observe later, working in FM-domain theory is equivalent to working in a certain category of functors.

4.1 FM-sets, FM-cpos and FM-cppos

For simplicity, we work with respect to a fixed countably infinite set $\mathbb{A}$ of *atoms*. The theory could, however, be generalised to handle different *sorts* of atoms rather like one sees in Fresh O'Caml and in *FMS-set theory*[20].

Definition 4.1.1 (Permutations and actions). Let a *permutation* π be a member of the permutation group $perm(\mathbb{A})$ on $\mathbb{A}$. Such a π is therefore a bijection from $\mathbb{A}$ to itself. Write id for the identity permutation and $\pi' \circ \pi$ for that permutation acting as π and then as π'.

An *action* of the group $perm(\mathbb{A})$ on a set D is specified by a binary operator $\cdot_D : perm(\mathbb{A}) \times D \to D$ satisfying:

$$id \cdot_D d = d \quad \text{and} \quad \pi \cdot_D (\pi' \cdot_D d) = (\pi \circ \pi') \cdot_D d$$

for all $d \in D$. Call a set equipped with such an action a $perm(\mathbb{A})$*-set* and omit the D subscript of $\cdot_D$ when the meaning is clear. $\diamond$

For example, the set of atoms $\mathbb{A}$ has a canonical permutation action $(\pi, a) \mapsto \pi(a)$. One particular set may determine various $perm(\mathbb{A})$-sets by varying the action: for example, we could construct another $perm(\mathbb{A})$-set based on $\mathbb{A}$ by equipping it with the null action $(\pi, a) \mapsto a$.

We shall often be concerned with those permutations π which consist of a single transposition of two atoms. Let us write $(a\ a')$ for that permutation exchanging a and a' and acting as the identity otherwise. The following lemma then follows immediately.

Lemma 4.1.2. *For atoms a, b, c, d then $(a\ b) \cdot (c\ d) \cdot x = (c\ d) \cdot ((c\ d) \cdot a\ (c\ d) \cdot b) \cdot x$ and $(a\ b) \cdot (c\ d) \cdot x = ((a\ b) \cdot c\ (a\ b) \cdot d) \cdot (a\ b) \cdot x$.* $\square$

Definition 4.1.3. For a finite set $\overline{a} \subseteq \mathbb{A}$ and a permutation π, say that π *fixes* $\overline{a}$ *pointwise* iff for all $a \in \overline{a}$, $\pi(a) = a$. $\diamond$

Definition 4.1.4 (Finite support). Let D be a $perm(\mathbb{A})$-set and take some $d \in D$. Say that d is *finitely supported* (with respect to the action $\cdot_D$) iff there exists a finite set $\overline{a} \subseteq \mathbb{A}$ such that each permutation π fixing $\overline{a}$ pointwise also fixes d (that is to say, $\pi \cdot d = d$).

It is a consequence of this definition that each finitely-supported element d of a $perm(\mathbb{A})$-set possesses a *least* finite support[20, Proposition 3.4]. Let us write $\mathrm{supp}(d)$ for this least finite support. When talking about multiple elements of some $perm(\mathbb{A})$-set(s) at one time, we write $\mathrm{supp}(d, d')$ etc. for the union of their supports in the obvious manner. It is in fact possible to express the support of an element $d \in D$ as an exact formula, thus:

$$\mathrm{supp}(d) \stackrel{\mathrm{def}}{=} \{a \in \mathbb{A} \mid \{b \in \mathbb{A} \mid (a\ b) \cdot d \neq d\} \text{ is not finite}\}$$

although this alternative definition is somewhat difficult to work with.

The support of an element d may be thought of as 'all of the atoms involved in d's construction'. This corresponds with the fact that for atoms a, a' not in the support of d then $(a\ a') \cdot d = d$. We say that such atoms are *fresh* for d, sometimes written $a \# d$.

By imposing the condition that each element of a $perm(\mathbb{A})$-set must be finitely supported, we arrive at the following definition.

Definition 4.1.5 (FM-sets). Let D be a $perm(\mathbb{A})$-set. Then D forms an *FM-set* iff all of its elements are finitely supported. $\diamond$

Definition 4.1.6 (Finitely supported and equivariant subsets). Given an FM-set D, say that $S \subseteq D$ is *finitely supported* iff there exists a finite set of atoms $\overline{a}$ such that for all π which fix $\overline{a}$ pointwise, $\{\pi \cdot s \mid s \in S\} = S$. Say that S is an *equivariant* subset if we can take $\overline{a}$ to be empty. Write $S \subseteq_{\mathrm{eq}} D$ just when S is an equivariant subset of D. $\diamond$

Definition 4.1.7 (Powersets). Given an FM-set D write $\mathcal{P}(D)$ for the *powerset* of D, whose members are all finitely-supported subsets of D. This becomes an FM-set when equipped with the action $\pi \cdot S \stackrel{\mathrm{def}}{=} \{\pi \cdot s \mid s \in S\}$ for all $S \subseteq D$. $\diamond$

Remark 4.1.8. Note that given FM-sets S, D with $S \subseteq D$ then we also have that $\pi \cdot S \subseteq \pi \cdot D$ for any permutation π. $\diamond$

Definition 4.1.9 (Products). Given FM-sets D and E, we can form their *product* $D \times E$ whose underlying set is that of the usual Cartesian product. Writing (d, e) for an element of $D \times E$, the permutation action is given by $\pi \cdot (d, e) \stackrel{\mathrm{def}}{=} (\pi \cdot_D d, \pi \cdot_E e)$. Each element of $D \times E$ is clearly finitely supported by virtue of the elements of D and E having this property. $\diamond$

Definition 4.1.10 (FM-relations). Say that a subset $R \subseteq D \times E$ is an *FM-relation* if it is a finitely-supported subset of $D \times E$ in the sense of Definition 4.1.6. $\diamond$

Lemma 4.1.11. *Given some FM-set D and any $d \in D$ then for all permutations π, $\mathrm{supp}(\pi \cdot d) \subseteq \pi \cdot \mathrm{supp}(d)$ (where the permutation action on the finite set of atoms $\mathrm{supp}(d)$ is as in Definition 4.1.7).*

Proof. We show that $\pi \cdot \mathrm{supp}(d)$ is a finite support (but not necessarily the least such) for $\pi \cdot d$. Given any permutation π' which fixes $\pi \cdot \mathrm{supp}(d) \stackrel{\mathrm{def}}{=} \{\pi(a) \mid a \in \mathrm{supp}(d)\}$ pointwise, we have that for all $a \in \mathrm{supp}(d)$, $\pi' \circ \pi \cdot a = \pi \cdot a$. Therefore $\pi^{-1} \circ \pi' \circ \pi \cdot a = a$. We need to show that $\pi' \circ \pi \cdot d = \pi \cdot d$, so it suffices to show $\pi^{-1} \circ \pi' \circ \pi \cdot d = d$. This follows from above where we see that $\pi^{-1} \circ \pi' \circ \pi$ fixes the support of d. $\square$

Lemma 4.1.12. *Given some FM-set D and any $d \in D$ then for all permutations π, $\mathrm{supp}(\pi \cdot d) = \pi \cdot \mathrm{supp}(d)$.*

Proof. Lemma 4.1.11 tells us that $\mathrm{supp}(\pi \cdot d) \subseteq \pi \cdot \mathrm{supp}(d)$. To get the other direction, we can instantiate the same Lemma using the permutation π^{-1} and the element $\pi \cdot d \in \pi \cdot D$ to deduce that $\mathrm{supp}(\pi^{-1} \cdot \pi \cdot d) \subseteq \pi^{-1} \cdot \mathrm{supp}(\pi \cdot d)$. It follows that $\pi \cdot \mathrm{supp}(d) \subseteq \mathrm{supp}(\pi \cdot d)$ by equivariance of $\subseteq$ (viz. Remark 4.1.8). $\square$

As is usual when giving a denotational semantics to a programming language we shall require a mathematical object with more structure than a set in order to denote function types. We use an analogue of the familiar notion of chain-complete partially-ordered set (cpo) as follows.

Definition 4.1.13 (FM-cpos). An *FM-cpo* D is an FM-set equipped with a partial order relation $\sqsubseteq_D \subseteq D \times D$, written $d \sqsubseteq_D d'$ whenever $(d, d') \in \sqsubseteq_D$. We write $\sqsubseteq$ instead of $\sqsubseteq_D$ when the meaning is clear. The relation $\sqsubseteq$ must be reflexive, anti-symmetric, transitive and *equivariant*, that is to say 'must be closed under swapping':

$$d \sqsubseteq d' \Rightarrow \forall \pi \in perm(\mathbb{A}).\ \pi \cdot d \sqsubseteq \pi \cdot d'. \tag{4.1}$$

Furthermore, we require that any countable ascending chain $(d_n \mid n < \omega) \in D$ *which overall possesses a finite support* S (that is to say, $\mathrm{supp}(d_n) \subseteq S$ for all $n < \omega$) has a least upper bound $\bigsqcup_{n<\omega} d_n$. $\diamond$

Definition 4.1.14 (FM-complete lattices). An *FM-complete lattice* is an FM-set D equipped with an equivariant partial order relation $\sqsubseteq$ such that every *finitely-supported* subset $S \subseteq D$ has a greatest lower bound (and hence also a least upper bound). $\diamond$

Notation 4.1.15. Given a finitely-supported chain $(d_n \mid n < \omega)$ in some FM-cpo D, write $\mathrm{supp}\,(d_n \mid n < \omega)$ for the least finite support of the chain (so that $\mathrm{supp}(d_n) \subseteq \mathrm{supp}\,(d_n \mid n < \omega)$ for all $n < \omega$). $\diamond$

Note that an FM-cpo may not have least upper bounds of *all* countable ascending chains. For example, consider the FM-cpo whose underlying set consists of all finite subsets of $\mathbb{A}$ ordered by inclusion. Enumerating[1] the elements of $\mathbb{A}$ as $a_0, a_1, \ldots$ then we get a chain $\{a_0\} \sqsubseteq \{a_0, a_1\} \sqsubseteq \cdots$ which is clearly not finitely supported and indeed does not possess a least upper bound. Indeed, any chain in this FM-cpo possesses finite support just when the chain is eventually constant.

In this thesis we will predominantly work with *pointed* FM-cpos called *FM-cppos*, defined as follows.

Definition 4.1.16 (FM-cppos). An FM-cppo is an FM-cpo D equipped with a distinguished least element $\bot$: for all $d \in D$, $\bot \sqsubseteq d$. Condition (4.1) then forces the support of $\bot$ to be empty in order for $\bot$ to be unique. $\diamond$

We have made the choice of using FM-cppos rather than FM-cpos just because the many and various domain-theoretic constructions seem to work rather more smoothly in this setting. Morally speaking too, there is something more pleasant about FM-cppos in general. For example, we shall see shortly how to construct a strict function space $D \multimap D'$ between FM-cppos; by virtue of using pointed sets the unlift of such a construction is immediately well-defined without having even to consider the pointedness of D' as one would otherwise.

Lemma 4.1.17. *Let D be an FM-cpo. Then for a finitely-supported chain $(d_n \mid n < \omega)$ in D, we have $\pi \cdot \bigsqcup_{n<\omega} d_n = \bigsqcup_{n<\omega} \pi \cdot d_n$.*

Proof. Follows immediately from (4.1), since this shows that the bijection π preserves $\sqsubseteq_D$. $\square$

Lemma 4.1.18. *Take a chain $(d_n \mid n < \omega)$ in an FM-cpo D. Then we have $\mathrm{supp}(\bigsqcup_{n<\omega} d_n) \subseteq \mathrm{supp}\,(d_n \mid n < \omega)$.*

Proof. We proceed by showing that all π fixing $\mathrm{supp}\,(d_n \mid n < \omega)$ pointwise also fix $\bigsqcup_{n<\omega} d_n$, for then we know that $\mathrm{supp}\,(d_n \mid n < \omega)$ is a finite support for $\bigsqcup_{n<\omega} d_n$. For such a π then it is clear that $\pi \cdot d_n = d_n$. Therefore $\bigsqcup_{n<\omega} \pi \cdot d_n = \bigsqcup_{n<\omega} d_n$ and we can conclude by Lemma 4.1.17. $\square$

4.2 Some FM-cpos and their construction

4.2.1 Atoms, lifting, products and sums

Definition 4.2.1 (The FM-cpo of atoms). The set $\mathbb{A}$ can be turned into an FM-cpo (also confusingly named $\mathbb{A}$) specified as follows.

▷ *Underlying set:* the set of atoms $\mathbb{A}$.

▷ *Permutation action:* canonical action $\pi \cdot_{\mathbb{A}} a \stackrel{\mathrm{def}}{=} \pi(a)$ for all $a \in \mathbb{A}$.

▷ *Partial order:* $a \sqsubseteq_{\mathbb{A}} a' \stackrel{\mathrm{def}}{\Leftrightarrow} a = a'$, for all $a, a' \in \mathbb{A}$. $\diamond$

[1] Such a construction could not be performed formally inside FM-set theory, as the bijection does not possess finite support.

It is a consequence of this definition that $\mathrm{supp}(a) = \{a\}$ for all $a \in \mathbb{A}$.

Definition 4.2.2 (Lifting). The *lift* of an FM-cpo D, written $D_\perp$ is the FM-cppo specified by the following.
▷ *Underlying set:* is given by $D \uplus \{\perp\}$ (note that we assume $\perp \notin D$).
▷ *Permutation action:* is as for D.
▷ *Partial order:* $\{(d, d') \in (D \cup \{\perp\}) \times (D \cup \{\perp\}) \mid d \sqsubseteq_D d' \ \vee \ d = \perp\}$.
▷ *Least element:* $\perp$. $\diamond$

Definition 4.2.3 (Unlifting). The *unlift* of an FM-cppo D, written $D_{\downarrow}$ is the FM-cpo specified by the following.
▷ *Underlying set:* the underlying set of D with $\perp$ removed.
▷ *Permutation action:* is as for[2] D.
▷ *Partial order:* $\{(d, d') \in D \times D \mid d \sqsubseteq_D d' \ \wedge \ d \neq \perp\}$. $\diamond$

It is a consequence of these definitions that the least finite support of a non-bottom element $d \in D$, for some FM-cppo D, is the same as that of the element $d \in D_{\downarrow}$. Similarly, given an element e of an FM-cpo E then the least finite support of e is equal to that of the element $e \in E_\perp$.

Definition 4.2.4 (Products). The *product* of FM-cpos D and E, written $D \times E$ is the FM-cpo specified by the following.
▷ *Underlying set:* $\{(d, e) \mid d \in D \ \wedge \ e \in E\}$.
▷ *Permutation action:* $\pi \cdot_{D\times E} (d, e) \stackrel{\text{def}}{=} (\pi \cdot_D d, \pi \cdot_E e)$.
▷ *Partial order:* $(d, e) \sqsubseteq_{D\times E} (d', e') \stackrel{\text{def}}{\Leftrightarrow} d \sqsubseteq_D d' \ \wedge \ e \sqsubseteq_E e'$. $\diamond$

It follows that given FM-cpos D and E, the least finite support of an element $(d, e) \in D \times E$ is given by $\mathrm{supp}(d) \cup \mathrm{supp}(e)$. Furthermore, if D and E are in fact FM-cppos then so is $D \times E$, with least element $(\perp_D, \perp_E)$.

Definition 4.2.5 (Smash products). The *smash product* of FM-cppos D and E is the FM-cppo $D \otimes E \stackrel{\text{def}}{=} (D_{\downarrow} \times E_{\downarrow})_\perp$. $\diamond$

It follows that given FM-cppos D, E the least finite support of a non-bottom element, written $\langle d, e\rangle \in D \otimes E$, is $\mathrm{supp}(d) \cup \mathrm{supp}(e)$.

Definition 4.2.6 (Coalesced sums). The *coalesced sum* $D \oplus E$ of FM-cppos D and E is specified by the following.
▷ *Underlying set:* $\{\perp_{D\oplus E}\} \cup \{\mathrm{in}_1(d) \mid d \in D \ \wedge \ d \neq \perp_D\} \cup \{\mathrm{in}_2(e) \mid e \in E \ \wedge \ e \neq \perp_E\}$, where $\perp_{D\oplus E}$ is a distinguished least element.
▷ *Permutation action:* defined as follows for $s \in D \oplus E$:

$$\pi \cdot s \stackrel{\text{def}}{=} \begin{cases} \mathrm{in}_1(\pi \cdot_D d) & \text{if } s = \mathrm{in}_1(d); \\ \mathrm{in}_2(\pi \cdot_E e) & \text{if } s = \mathrm{in}_2(e); \\ \perp_{D\oplus E} & \text{otherwise.} \end{cases}$$

▷ *Partial order:* is again defined by case analysis:

$$\begin{array}{ll} \perp_{D\oplus E} \sqsubseteq d & \text{for all } d \in D \oplus E; \\ \mathrm{in}_i(d) \sqsubseteq \mathrm{in}_j(d') & \text{iff } i = j \text{ and } d \sqsubseteq d'. \quad \diamond \end{array}$$

It follows that the least finite support of a non-bottom element $s \in D \oplus E$ is $\mathrm{supp}(d)$ if $s = \mathrm{in}_1(d)$; and $\mathrm{supp}(e)$ if $s = \mathrm{in}_2(e)$.

4.2.2 Functions and function spaces

Definition 4.2.7 (Permutation action for functions). Take FM-sets X and Y and let f be a total function from the underlying set of X to that of Y. We can equip f with a permutation action by defining

$$(\pi \cdot f)(x) \stackrel{\text{def}}{=} \pi \cdot_Y (f(\pi^{-1} \cdot_X x)). \quad \diamond \tag{4.2}$$

[2] Since $\pi \cdot_D \perp = \perp$ and π is a bijection on D then the action $\pi \cdot_D$ – does indeed induce a bijection on $D_{\downarrow}$.

Definition 4.2.8 (Finitely-supported functions). Say that f is a *finitely-supported function* iff it possesses a finite support with respect to the permutation action (4.2). Write $(X \to Y)$ for the FM-set of all functions from X to Y which are finitely supported with respect to the above permutation action. $\diamond$

The construction of the FM-set $(X \to Y)$ highlights the main difference between FM-domain theory and classical domain theory: in the FM world, we must be careful when forming subsets and function spaces to just ensure that we remain within the world of objects with finite support.

Definition 4.2.9 (Equivariant functions). For FM-sets X, Y say that $f \in (X \to Y)$ is *equivariant* iff for all permutations π, $\pi \cdot f = f$. $\diamond$

It is a consequence of this definition that f is equivariant just when for all permutations π and all $x \in \mathrm{dom}(f)$, $f(\pi \cdot x) = \pi \cdot (f(x))$. We may also refer to an equivariant function as 'having empty support', since clearly f is equivariant just when $\mathrm{supp}(f) = \emptyset$.

Definition 4.2.10 (Monotone and continuous functions). Take FM-cpos X and Y and a finitely-supported function f from the underlying FM-set of X to that of Y. Say f is *monotone* iff $x \sqsubseteq_X x'$ implies that $f(x) \sqsubseteq_Y f(x')$. Say f is *continuous* iff it is monotone and for each finitely-supported chain $(x_n \mid n < \omega)$ in X, $f(\bigsqcup_{n<\omega} x_n) = \bigsqcup_{n<\omega}(f(x_n))$. $\diamond$

Definition 4.2.11 (Strict functions). Given FM-cppos X, Y and a finitely-supported function f from the underlying FM-set of X to that of Y, say that f is *strict* iff $f(\bot_X) = \bot_Y$. $\diamond$

Definition 4.2.12 (Strict continuous function spaces). Given FM-cppos X and Y write $X \multimap Y$ for that FM-cppo specified as follows.
▷ *Underlying set:* all total, finitely-supported functions $f \in (X \to Y)$ which are both strict and continuous.
▷ *Permutation action:* is as for $(X \to Y)$.
▷ *Partial order:* is pointwise: $f \sqsubseteq_{X \multimap Y} f'$ iff for all $x \in X$, $f(x) \sqsubseteq_Y f'(x)$.
▷ *Least element:* the constantly-bottom function. $\diamond$

Remark 4.2.13. It must be remembered that an FM-cppo $X \multimap Y$ may well contain many functions which are not equivariant. For example, consider the FM-cppo $1_\bot$ whose underlying set consists of $\{\bot, \top\}$ with $\bot \sqsubseteq \top$, $\bot \sqsubseteq \bot$, $\top \sqsubseteq \top$, $\pi \cdot \bot \stackrel{\mathrm{def}}{=} \bot$ and $\pi \cdot \top \stackrel{\mathrm{def}}{=} \top$. This may be used to form the FM-cppo $1_\bot \multimap \mathbb{A}_\bot$ whose elements will include functions mapping $\top$ to some $a \in \mathbb{A}$. Call such a function f. This function cannot have empty support, because if it were to then for any permutation π and $x \in 1_\bot$ we would have $\pi \cdot (f(x)) = f(\pi \cdot x) = f(x)$. This cannot hold since we could set π to be any permutation exchanging a with some different atom. $\diamond$

Definition 4.2.14 (Dependent products). The dependent product of FM-cppos D_i indexed by a set I is the FM-cppo written as $\prod_{i \in I} D_i$ and specified by the following.
▷ *Underlying set:* All finitely-supported functions ρ with domain I such that for all $i \in I$, $\rho(i)$ lies in D_i.
▷ *Permutation action:* is given by $(\pi \cdot \rho)(i) \stackrel{\mathrm{def}}{=} \pi \cdot_{D_i} (\rho(i))$.
▷ *Partial order:* is pointwise: $\rho \sqsubseteq \rho'$ iff for all $i \in I$, $\rho(i) \sqsubseteq_{D_i} \rho'(i)$.
▷ *Least element:* That function mapping each $i \in I$ to the least element of D_i. $\diamond$

Definition 4.2.15 (Dependent smash products). The dependent smash product of FM-cppos D_i indexed by a set I is the FM-cppo $\bigotimes_{i \in I} D_i$ specified as follows:
▷ *Underlying set:* All finitely-supported functions ρ with domain I such that for all $i \in I$, $\rho(i)$ lies in $D_i \setminus \{\bot\}$; together with the least element (see below).
▷ *Permutation action:* is given by $(\pi \cdot \rho(i)) \stackrel{\mathrm{def}}{=} \pi \cdot_{D_i} (\rho(i))$.
▷ *Partial order:* is pointwise: $\rho \sqsubseteq \rho'$ iff for all $i \in I$, $\rho(i) \sqsubseteq_{D_i} \rho'(i)$.
▷ *Least element:* That function mapping each $i \in I$ to the least element of D_i. $\diamond$

Note that we place no restriction on the cardinality of the index set I when constructing a dependent product or dependent smash product. One must simply take care to ensure that each member of such a product is a *finitely-supported* function out of I.

Two familiar notions which we shall require in §4.5 are those of *embeddings* and *projections*, defined as follows.

Definition 4.2.16 (Embeddings and projections). An *embedding* between FM-cppos is a function $e \in D \multimap E$ for which there exists a (necessarily uniquely-determined) second function $p \in E \multimap D$, known as a *projection*, such that $p \circ e = id_D$ and $e \circ p \sqsubseteq id_E$. $\diamond$

4.2.3 Abstraction FM-cppos

We now come to the main novel construction of this section, that of *abstraction FM-cppos*. In order to perform this construction, we shall first establish some preliminary results.

Definition 4.2.17. For each FM-cppo D, define the relation $\preceq_D \subseteq (\mathbb{A} \times D) \times (\mathbb{A} \times D)$ as follows:

$$(a, d) \preceq_D (a', d') \stackrel{\text{def}}{\Leftrightarrow} \exists a'' \in \mathbb{A} \setminus \mathrm{supp}(a, a', d, d'). \ (a\ a'') \cdot d \sqsubseteq_D (a'\ a'') \cdot d'.$$

We write $\preceq$ rather than $\preceq_D$ when the meaning is clear. $\diamond$

We aim to show that $\preceq$ is a pre-order. To do this, we require a lemma which establishes another example of the key 'some/any' property first seen in §3.4.1.

Lemma 4.2.18. $(a, d) \preceq (a', d') \Leftrightarrow \forall a'' \in \mathbb{A} \setminus \mathrm{supp}(a, a', d, d'). \ (a\ a'') \cdot d \sqsubseteq_D (a'\ a'') \cdot d'$.

Proof. The reverse direction follows immediately, since $\mathbb{A} \setminus \mathrm{supp}(a, a', d, d')$ is cofinite. For the forwards direction, since $(a, d) \preceq (a', d')$ then there exists some $a'' \in \mathbb{A} \setminus \mathrm{supp}(a, a', d, d')$ such that $(a\ a'') \cdot d \sqsubseteq_D (a'\ a'') \cdot d'$. Now given any $a''' \in \mathbb{A} \setminus \mathrm{supp}(a, a', d, d')$ then the equivariance of $\sqsubseteq_D$ gives us that $(a''\ a''') \cdot (a\ a'') \cdot d \sqsubseteq_D (a''\ a''') \cdot (a'\ a'') \cdot d'$. But by virtue of Lemma 4.1.2 and the way we have selected a'' and a''', this implies that $(a\ a''') \cdot d \sqsubseteq_D (a'\ a''') \cdot d'$. $\square$

Lemma 4.2.19. *The relation $\preceq$ is a pre-order.*

Proof. It is clear that $\preceq$ is reflexive since $\sqsubseteq_D$ is too. To see transitivity, suppose $(a, d) \preceq (a', d')$ and $(a', d') \preceq (a'', d'')$. Then there exist atoms a_1, a_2 such that $(a\ a_1) \cdot d \sqsubseteq_D (a'\ a_1) \cdot d'$ and $(a'\ a_2) \cdot d' \sqsubseteq_D (a''\ a_2) \cdot d''$ with $a_1 \in \mathbb{A} \setminus \mathrm{supp}(a, a', d, d')$ and $a_2 \in \mathbb{A} \setminus \mathrm{supp}(a', a'', d', d'')$. But by Lemma 4.2.18 we can deduce that for $a_3 \in \mathbb{A} \setminus (a, a', a'', d, d', d'')$, $(a\ a_3) \cdot d \sqsubseteq_D (a'\ a_3) \cdot d'$ and $(a'\ a_3) \cdot d' \sqsubseteq_D (a''\ a_3) \cdot d''$. We can now conclude immediately by transitivity of $\sqsubseteq_D$. $\square$

Definition 4.2.20. Given an FM-cppo D, define the relation $\sim \subseteq (\mathbb{A} \times D) \times (\mathbb{A} \times D)$ as follows:

$$(a, d) \sim (a', d') \stackrel{\text{def}}{\Leftrightarrow} (a, d) \preceq (a', d') \ \wedge \ (a', d') \preceq (a, d). \quad \diamond \tag{4.3}$$

It follows from Lemma 4.2.19 that $\sim$ is an equivalence relation; the pre-order $\preceq$ induces a partial order on equivalence classes $(a, d)/\!\sim$. Let us show that such equivalence classes are closed under permuting atoms.

Lemma 4.2.21 ($\preceq$ is equivariant). *$(a, d) \preceq (a', d')$ implies that for all permutations π, $(\pi(a), \pi \cdot_D d) \preceq (\pi(a'), \pi \cdot_D d')$.*

Proof. Assume $(a, d) \preceq (a', d')$. That is to say, for some (or indeed any) $a'' \in \mathbb{A} \setminus \mathrm{supp}(a, a', d, d')$ then $(a\ a'') \cdot d \sqsubseteq_D (a'\ a'') \cdot d'$. Given any permutation π the equivariance property (4.1) of $\sqsubseteq_D$ implies that $(\pi(a)\ \pi(a'')) \cdot (\pi \cdot_D d) \sqsubseteq_D (\pi(a')\ \pi(a'')) \cdot (\pi \cdot_D d')$. This enables us to conclude that $\pi \cdot (a, d) \preceq \pi \cdot (a', d')$ so long as $\pi(a'')$ lies in $\mathbb{A} \setminus \mathrm{supp}(\pi(a), \pi(a'), \pi \cdot_D d, \pi \cdot_D d')$. But this follows by virtue of the condition on a''. $\square$

Corollary 4.2.22. *The equivalence relation $\sim$ is equivariant.*

Proof. Immediate from Lemma 4.2.21 and (4.3). $\square$

We are now in a position to introduce the following important definition, which forms one of the main novelties of this chapter.

Definition 4.2.23 (Abstraction FM-cppos). For any FM-cppo D, the *atom-abstraction* FM-cppo $[\mathbb{A}]D$ is specified by the following.

▷ *Underlying set:* the set of equivalence classes $(a, d)/\!\sim$ in $\mathbb{A} \times D$. Let us write $[a]d$ for one such equivalence class.

▷ *Permutation action:* $\pi \cdot_{[\mathbb{A}]D} [a]d \stackrel{\text{def}}{=} [\pi(a)]\pi \cdot_D D$. This correctly preserves equivalence classes by virtue of Lemma 4.2.22.

▷ *Partial order:* $[a]d \sqsubseteq_{[\mathbb{A}]D} [a']d' \stackrel{\text{def}}{\Leftrightarrow} (a, d) \preceq (a', d')$.

▷ *Least element:* the equivalence class $[a]\bot_D$, for any $a \in \mathbb{A}$. Note that given any other $a' \in \mathbb{A}$ then we have $[a]\bot_D \sim [a']\bot_D$. ⋄

We can think of elements $[a]d \in [\mathbb{A}]D$ as representing elements of D with one atom abstracted (bound); furthermore, these are identified up to 'renaming of the bound atom'.

We must now proceed to verify that the construction in the definition above does indeed satisfy the requirements for an FM-cppo. To do this, we shall first require some additional lemmata.

Lemma 4.2.24. *$[a]d = [a']d'$ in $[\mathbb{A}]D$ just when*[3] *either $a = a' \wedge d = d'$ or $a \neq a' \wedge a \notin \mathrm{supp}(d') \wedge d = (a\ a') \cdot d'$.*

Proof. In the forwards direction, assume $[a]d = [a']d'$. That is to say, for all $a'' \in \mathbb{A} \setminus \mathrm{supp}(a, a', d, d')$ then $(a\ a'') \cdot d = (a'\ a'') \cdot d'$ in D. Now proceed by case analysis. If $a = a'$ then $(a\ a'') \cdot d = (a'\ a'') \cdot d'$ and so $d = d'$ by equivariance of $\sqsubseteq_D$. For the other case, take $a \neq a'$. To see that $a \notin \mathrm{supp}(d')$, take any $a'' \notin \mathrm{supp}(a, a', d, d')$. Then $a = (a'\ a'') \cdot a = (a'\ a'') \cdot (a\ a'') \cdot a''$. Since $a'' \notin \mathrm{supp}(d)$, Lemma 4.1.12 implies that $(a'\ a'') \cdot (a\ a'') \cdot a'' \notin \mathrm{supp}((a'\ a'') \cdot (a\ a'') \cdot d)$. But we also have that $(a\ a'') \cdot d = (a'\ a'') \cdot d'$, so $a \notin \mathrm{supp}((a'\ a'') \cdot (a'\ a'') \cdot d') = \mathrm{supp}(d')$. Then $(a\ a'') \cdot d = (a'\ a'') \cdot d'$ implies $(a\ a'') \cdot (a\ a'') \cdot d = (a\ a'') \cdot (a'\ a'') \cdot d'$, so $d = ((a\ a'') \cdot a'\ (a\ a'') \cdot a'') \cdot (a\ a'') \cdot d' = (a'\ a) \cdot (a\ a'') \cdot d' = (a\ a') \cdot d'$ (since $a \notin \mathrm{supp}(d')$).

In the reverse direction, take atoms a, a' and proceed again by case analysis. When $a = a'$ and $d = d'$, the result follows easily by equivariance of $\sqsubseteq_D$. For the other case, take $a \neq a'$, $a \notin \mathrm{supp}(d')$ and $d = (a\ a') \cdot d'$. Taking any $a'' \in \mathbb{A} \setminus (a, a', d, d')$ we have that $(a\ a'') \cdot d = (a\ a'') \cdot (a\ a') \cdot d' = (a''\ a') \cdot d'$, which implies $[a]d = [a']d'$. □

Lemma 4.2.25. *$d \sqsubseteq_D d'$ holds just when for any atom a, $[a]d \sqsubseteq_{[\mathbb{A}]D} [a]d'$.*

Proof. Given $d \sqsubseteq_D d'$ then we can always pick an atom $a' \in \mathbb{A} \setminus \mathrm{supp}(d, d')$. The result then follows from the equivariance of $\sqsubseteq_D$. A similar argument applies in the reverse direction. □

Lemma 4.2.26. *For an element $[a]d$ in $[\mathbb{A}]D$, then $\mathrm{supp}([a]d) \subseteq \mathrm{supp}(d) \setminus \{a\}$.*

Proof. We show that $\mathrm{supp}(d) \setminus \{a\}$ is a finite support (but not necessarily the least such) for $[a]d$. Take any permutation π which fixes $\mathrm{supp}(d) \setminus \{a\}$ pointwise. We wish to know that $\pi \cdot [a]d = [\pi(a)]\pi \cdot d = [a]d$. By Lemma 4.2.24 it therefore suffices to show that either $a = \pi(a) \wedge d = \pi \cdot d$; or $a \neq \pi(a) \wedge a \notin \mathrm{supp}(\pi \cdot d) \wedge d = (a\ \pi(a)) \cdot (\pi \cdot d)$. If $a = \pi(a)$, it follows that π fixes $\mathrm{supp}(d)$ pointwise and therefore $\pi \cdot d = d$; we have the result since in this case $(a\ \pi(a))$ is the identity.

In the case where $a \neq \pi(a)$, we first wish to know that $a \notin \mathrm{supp}(\pi \cdot d)$. Lemma 4.1.12 tells us that it suffices to prove $a \notin \{\pi(a') \mid a' \in \mathrm{supp}(d)\}$. If this did not hold, then we would have $a \in \{\pi(a') \mid a' \in \mathrm{supp}(d)\}$. Then there would exist some $a' \in \mathrm{supp}(d)$ such that $a = \pi(a')$; moreover, since $a \neq \pi(a)$ (and π is injective) then this implies $a \neq a'$. It follows that $a' \in \mathrm{supp}(d) \setminus \{a\}$ and so $\pi(a') = a'$. But then $a = \pi(a') = a'$, which would contradict our assumption that $a \neq a'$.

To complete the proof, we just need to show that in the $a \neq \pi(a)$ case, $d = (a\ \pi(a)) \cdot (\pi \cdot d)$. We do this by showing that the permutation $(a\ \pi(a)) \cdot \pi$ fixes all $a' \in \mathrm{supp}(d)$ (for if it does this, it must fix d also). Proceed by case analysis on a'. If $a' = a$ then the permutation is clearly the identity. Otherwise, a' is mapped to $\pi(a')$ by the permutation π. Since π fixes all atoms in $\mathrm{supp}(d) \setminus \{a\}$, then $\pi(a') = a'$ (and this is not equal to a). Moreover, $\pi(a)$ cannot equal $\pi(a')$ since $a \neq a'$ and π is injective. Therefore $(a\ \pi(a)) \cdot \pi$ maps a' to itself, as required. □

Lemma 4.2.27 (Supports in $[\mathbb{A}]D$). *Let $[a]d$ be a member of $[\mathbb{A}]D$. Then $\mathrm{supp}([a]d) = \mathrm{supp}(d) \setminus \{a\}$.*

[3] Do not be fooled into thinking that this lemma generalises to the case of the partial order relation $\sqsubseteq$ rather than just equality, for it does not. We explain why in §4.2.4.

Proof. By Lemma 4.2.26 it remains to show that $\mathrm{supp}([a]d)$ is not less than $\mathrm{supp}(d) \setminus \{a\}$. Without loss of generality suppose that $\mathrm{supp}(d) \setminus \{a, a'\}$ supports $[a]d$, for some a' not equal to a and occurring in $\mathrm{supp}(d)$. Now consider the permutation π which just exchanges a and a', thus fixing $\mathrm{supp}(d) \setminus \{a, a'\}$ pointwise. It follows that $\pi \cdot [a]d = [a'](a\ a') \cdot d$ and because $\mathrm{supp}(d) \setminus \{a, a'\}$ supports $[a]d$, we must have $[a'](a\ a') \cdot d = [a]d$. Applying Lemma 4.2.24, it follows that $a' \notin \mathrm{supp}(d)$. But this contradicts the assumption above that a' occurs in $\mathrm{supp}(d)$. □

Lemma 4.2.28 (Concretion). *For each $[a]d$ in $[\mathbb{A}]D$ and $a' \in \mathbb{A} \setminus \mathrm{supp}([a]d)$, then there exists a unique $d' = (a\ a') \cdot d$ such that $[a]d = [a']d'$. We call this d' the concretion of $[a]d$ at a' and write it $([a]d) @ a'$.*

Proof. Since $a' \in \mathbb{A} \setminus \mathrm{supp}([a]d)$ then Lemma 4.2.27 tells us that either $a = a'$, or $a \neq a'$ and $a' \notin \mathrm{supp}(d)$. In the first case, d' may be specified just by $d = (a\ a') \cdot d$, meaning that $[a]d = [a']d'$. Lemma 4.2.24 tells us that (in this case) the equality only holds for this particular d' and so it is unique. In the case where $a \neq a'$ and $a' \notin \mathrm{supp}(d)$, we again appeal to Lemma 4.2.24 to deduce that $d' = (a\ a') \cdot d$ is the unique d' such that $[a]d = [a']d'$. □

Lemma 4.2.29 (Supports of concretions). *For $a' \notin \mathrm{supp}([a]d)$ then we have $\mathrm{supp}(([a]d) @ a') \subseteq \mathrm{supp}([a]d) \cup \{a'\}$.*

Proof. We wish to calculate $\mathrm{supp}((a\ a') \cdot d)$. Since $a' \notin \mathrm{supp}([a]d)$ then Lemma 4.2.27 tells us that either $a = a'$, or $a \neq a'$ and $a' \notin \mathrm{supp}(d)$. If $a = a'$ then $\mathrm{supp}((a\ a') \cdot d) = \mathrm{supp}(d) = \mathrm{supp}([a]d) \cup \{a\}$ as required. If $a \neq a'$ and $a' \notin \mathrm{supp}(d)$ then by inspection we see that $\mathrm{supp}((a\ a') \cdot d)$ cannot contain a, but may contain a'. Therefore $\mathrm{supp}((a\ a') \cdot d) \subseteq \mathrm{supp}(d) \setminus \{a\} \cup \{a'\} = \mathrm{supp}([a]d) \cup \{a'\}$. □

Lemma 4.2.30 (Concretion order-preserving). *If $[a]d \sqsubseteq_{[\mathbb{A}]D} [a']d'$ then $([a]d) @ a'' \sqsubseteq_D ([a']d') @ a''$, for any $a'' \in \mathbb{A} \setminus \mathrm{supp}([a]d, [a']d')$.*

Proof. By Lemma 4.2.28 we know that $[a]d = [a''](([a]d) @ a'') = [a''](a\ a'') \cdot d$ and $[a']d = [a'']([a']d @ a'') = [a''](a'\ a'') \cdot d'$. Therefore $[a''](a\ a'') \cdot d \sqsubseteq_{[\mathbb{A}]D} [a''](a'\ a'') \cdot d'$ and so by Lemma 4.2.25 we have that $(a\ a'') \cdot d \sqsubseteq_D (a'\ a'') \cdot d'$ as required. □

Lemma 4.2.31. *Given $[a]d$ in $[\mathbb{A}]D$ and $a' \neq a''$ with $a', a'' \notin \mathrm{supp}([a]d)$, then $(a'\ a'') \cdot (([a]d) @ a') = ([a]d) @ a''$.*

Proof. By Lemma 4.2.28 we need to show that $(a'\ a'') \cdot (a\ a') \cdot d = (a\ a'') \cdot d$. But the left-hand side is equal to $(a\ a') \cdot ((a\ a') \cdot a'\ (a\ a') \cdot a'') \cdot d$. This can be shown to be equal to the right-hand side by a tedious case analysis, whose details we omit. □

Lemma 4.2.32. *Any finitely-supported chain $([a_n]d_n \mid n < \omega)$ in $[\mathbb{A}]D$ possesses a least upper bound which may be constructed as $[a] \bigsqcup_{n<\omega}(([a_n]d_n) @ a)$, for any $a \notin \mathrm{supp}\,([a_n]d_n \mid n < \omega)$.*

Proof. For all $n < \omega$, $a \notin \mathrm{supp}([a_n]d_n)$ and therefore we can concrete each element of the chain at a. By Lemma 4.2.30 this forms another chain in D, namely $(([a_n]d_n) @ a \mid n < \omega)$. This chain is also finitely supported, since by Lemma 4.2.29 we have that $\mathrm{supp}([a_n]d_n) @ a) \subseteq \mathrm{supp}([a_n]d_n) \cup \{a\}$ for all $n < \omega$. (The new chain is thus finitely supported by the set $\mathrm{supp}\,([a_n]d_n \mid n < \omega) \cup \{a\}$.) It follows that this chain has a least upper bound given by $\bigsqcup_{n<\omega}(([a_n]d_n) @ a)$ and for all $n < \omega$, $([a_n]d_n) @ a \sqsubseteq_D \bigsqcup_{n<\omega}(([a_n]d_n) @ a)$. It follows by Lemma 4.2.25 that $[a](([a_n]d_n) @ a) \sqsubseteq [a]\bigsqcup_{n<\omega}(([a_n]d_n) @ a)$ for all $n < \omega$; thus, $[a]\bigsqcup_{n<\omega}(([a_n]d_n) @ a)$ is an upper bound for the chain in $[\mathbb{A}]D$. To see that it is the least such, suppose that there is some other upper bound $[a']d'$ so that $[a_n]d_n \sqsubseteq [a']d'$ for all $n < \omega$. We wish to show that $[a]\bigsqcup_{n<\omega}(([a_n]d_n) @ a) \sqsubseteq [a']d'$, which we can do by choosing $a'' \in \mathbb{A} \setminus \mathrm{supp}(a, a', ([a_n]d_n \mid n < \omega))$ and showing that $(a\ a'') \cdot \bigsqcup_{n<\omega}(([a_n]d_n) @ a) \sqsubseteq (a'\ a'') \cdot d'$. The left-hand side of this is equal to $\bigsqcup_{n<\omega}((a\ a'') \cdot (([a_n]d_n) @ a)) = \bigsqcup_{n<\omega}(([a_n]d_n) @ a'')$ by Lemmata 4.1.17 and 4.2.31. Next, Lemma 4.2.30 together with the assumption that $[a']d'$ is an upper bound tells us that for all $n < \omega$, $([a_n]d_n) @ a'' \sqsubseteq_D ([a']d') @ a''$. In particular, we therefore have that $\bigsqcup_{n<\omega}(([a_n]d_n) @ a'') \sqsubseteq_D ([a']d') @ a''$. But Lemma 4.2.28 tells us that the right-hand side of this is equal to $(a'\ a'') \cdot d'$ as we require. □

Lemma 4.2.33. *$[\mathbb{A}]D$ is an FM-cppo.*

Proof. This splits into several parts, as follows.
► *Finite support property.* By Lemma 4.2.27.
► *Least upper bounds of finitely-supported chains.* By Lemma 4.2.32.
► *Equivariance of* $\sqsubseteq_{[\mathbb{A}]D}$. Follows by virtue of its definition and Lemma 4.2.21.
► *Pointedness.* The FM-cppo D possesses a least element $\bot$ which has empty support. It follows from Lemma 4.2.25 that the least element in $[\mathbb{A}]D$ is specified by $[a]\bot$ for any $a \in \mathbb{A}$.
$[\mathbb{A}]D$ therefore satisfies all of the requirements for an FM-cppo. □

4.2.4 Some curiosities

FM-domain theory—and abstraction FM-cppos in particular—contain some deep subtleties. Here we identify two traps for the unwary.

First, note that given some FM-cpo D and $d \sqsubseteq_D d'$ then we do not necessarily have that $\mathrm{supp}(d) \subseteq \mathrm{supp}(d')$. A simple example is given when D is the atom-abstraction FM-cpo $[\mathbb{A}]P$, where P is the FM-cpo with underlying FM-set $\mathcal{P}(\mathbb{A})$ (viz. Definition 4.1.7) and partially-ordered by inclusion. Choosing distinct atoms a_0, a_1, the fact that $\{a_0\} \subseteq (\mathbb{A} \setminus \{a_1\})$ means we can construct a chain in $[\mathbb{A}]P$ starting as follows:

$$[a_1]a_0 \sqsubseteq [a_1](\mathbb{A} \setminus \{a_1\}) \sqsubseteq \cdots$$

(This chain has an overall finite support, namely the set $\{a_0\}$—the crucial observation here being that the *cofiniteness* of $\mathbb{A} \setminus \{a_1\}$ implies that it is finitely supported by $\{a_1\}$.) We now have that

$$\mathrm{supp}([a_1]a_0) \supseteq \mathrm{supp}([a_1](\mathbb{A} \setminus \{a_1\})) \supseteq \cdots$$

which shows that the elements' support is not monotonically-increasing.

Our second example is the following false lemma—a version of Lemma 4.2.24 with $\sqsubseteq$ substituted for $=$.

Lemma 4.2.34 (False). *$[a]d \sqsubseteq [a']d'$ in $[\mathbb{A}]D$ just when either $a = a' \wedge d \sqsubseteq d'$ or $a \neq a' \wedge a \notin \mathrm{supp}(d') \wedge d \sqsubseteq (a\ a') \cdot d'$.* ⋄

To see an example of why this lemma fails to hold, take distinct atoms a_1, a_2, a_3 and construct the chain in $[\mathbb{A}]P$ starting as

$$[a_3]\{a_1\} \sqsubseteq [a_1](\mathbb{A} \setminus \{a_2, a_3\}) \sqsubseteq \cdots$$

We can now see that: $a_3 \neq a_1$; $a_3 \in \mathrm{supp}(\mathbb{A} \setminus \{a_2, a_3\})$ and moreover, since $(a_3\ a_1) \cdot (\mathbb{A} \setminus \{a_2, a_3\}) = \mathbb{A} \setminus \{a_1, a_2\}$ then $\{a_1\} \not\sqsubseteq (a_3\ a_1) \cdot \mathrm{supp}(\mathbb{A} \setminus \{a_2, a_3\})$—contradicting the False Lemma.

4.3 Fixed points

As is usual in denotational semantics, we shall require the notion of *fixed point* to give a meaning to recursively-defined functions. We will be needing two constructions of fixed points: that given by continuous functions on finitely-supported chains and that given by monotone functions on FM-complete lattices.

Definition 4.3.1. Given some FM-cppo D and a monotone function $f \in (D \to D)$, say that an element $d \in D$ is a *pre-fixed point* for f iff $f(d) \sqsubseteq d$. Say d is a *fixed point* for f iff $f(d) = d$. ⋄

Definition 4.3.2. Given an FM-cppo D and a function $f \in (D \to D)$, define $f^n(d)$ by induction on n as follows:

$$f^0(d) \stackrel{\mathrm{def}}{=} d \qquad f^{n+1}(d) \stackrel{\mathrm{def}}{=} f(f^n(d)).$$ ⋄

Lemma 4.3.3 (Fixed points on chains). *For an FM-cppo D, each continuous function $f \in (D \to D)$ possesses a least fixed point $\mathrm{fix}(f)$ which may be constructed as $\bigsqcup_{n<\omega} f^n(\bot)$, with $\mathrm{supp}(\mathrm{fix}(f)) \subseteq \mathrm{supp}(f)$.*

Proof. We must first show that $(f^n(\bot) \mid n < \omega)$ is a finitely-supported chain in D in order for a least upper bound to exist. To do this, let us prove by induction that $\mathrm{supp}(f^n(\bot)) \subseteq \mathrm{supp}(f)$ for arbitrary $n < \omega$. We always have $\mathrm{supp}(\bot) \subseteq \mathrm{supp}(f)$. Assuming that $\mathrm{supp}(f^n(\bot)) \subseteq \mathrm{supp}(f)$, we can prove $\mathrm{supp}(f^{n+1}(\bot)) \subseteq \mathrm{supp}(f)$ by showing that every permutation which

fixes $\mathrm{supp}(f)$ pointwise also fixes $f^{n+1}(\bot)$. Taking such a permutation π, observe that the assumption gives us that $\pi \cdot f^n(\bot) = f^n(\bot)$; therefore $f^n(\bot) = \pi^{-1} \cdot f^n(\bot)$. We also know that $\pi \cdot f = f$. It follows (using Definition 4.2.7) that $\pi \cdot f(f^n(\bot)) = (\pi \cdot f)(\pi^{-1} \cdot f^n(\bot)) = f(f^n(\bot)) = f^{n+1}(\bot)$ as required. Therefore $(f^n(\bot) \mid n < \omega)$ is finitely supported and indeed $\mathrm{supp}(\mathrm{fix}(f)) \subseteq \mathrm{supp}(f)$. That $\mathrm{fix}(f)$ is a least fixed point is a standard proof and we omit the remaining details. $\square$

The following lemma is an adaptation of the famous Tarski-Knäster fixed point theorem to the setting of FM-domain theory.

Lemma 4.3.4 (Fixed points on lattices). *Given an FM-complete lattice D, each monotone function $f \in (D \to D)$ possesses a least fixed point $\mathrm{fix}(f)$ which may be given as $\bigsqcap \{ d \in D \mid f(d) \sqsubseteq d \}$ (we use $\bigsqcap$ to indicate a greatest lower bound).*

Proof. The proof is standard and we omit the details, save for one crucial point. In order for the set $S \stackrel{\mathrm{def}}{=} \{ d \in D \mid f(d) \sqsubseteq d \}$ to have a greatest lower bound in the FM-complete lattice, it is necessary for it to be finitely supported. That this is so follows from the fact that $\mathrm{supp}(S) \subseteq \mathrm{supp}(f)$. To see this, take any permutation π which fixes $\mathrm{supp}(f)$ pointwise. We need to show that for all $s \in S$ then $\pi \cdot s \in S$. If $s \in S$ then $f(s) \sqsubseteq s$ and therefore $\pi \cdot f(s) \sqsubseteq \pi \cdot s$, which is equivalent to stating $(\pi \cdot f)(\pi \cdot s) \sqsubseteq \pi \cdot s$. Since π fixes f, then we have $f(\pi \cdot s) \sqsubseteq \pi \cdot s$ in D; thus, $\pi \cdot s \in S$. of S. $\square$

It is interesting to note that we can derive equivariance properties of fixed points without considering their explicit construction: the important property is that they are indeed fixed points. The following lemma highlights this: the result can be obtained not only by the proof given but also as a corollary of the proof of Lemma 4.3.4 above. We give this specific example since it is used in the next chapter to reason about an FM-complete lattice of relations.

Lemma 4.3.5. *Given an FM-complete lattice D and a monotone function $f \in (D \to D)$ with $\mathrm{supp}(f) = \emptyset$, then $\mathrm{supp}(\mathrm{fix}(f)) = \emptyset$.*

Proof. Take any atoms a, a' and observe $(a\ a') \cdot \mathrm{fix}(f) = (a\ a') \cdot (f(\mathrm{fix}(f))) = f((a\ a') \cdot \mathrm{fix}(f))$. Therefore $(a\ a') \cdot \mathrm{fix}(f)$ is a fixed point of f. It is also equal to $\mathrm{fix}(f)$, because $(a\ a') \cdot \mathrm{fix}(f) \sqsubseteq \mathrm{fix}(f)$ holds just when $\mathrm{fix}(f) \sqsubseteq (a\ a') \cdot \mathrm{fix}(f)$ by equivariance of $\sqsubseteq$. $\square$

4.4 Categorical constructions

Whilst we do not provide a categorical presentation of our denotational semantics, a modicum of category theory will be required in the next chapter. In order to do this, we now make several definitions which consist of transferring familiar categorical concepts into the FM setting.

Definition 4.4.1 (The category FM-Cpo). We take FM-cpos as the objects of the category; the morphisms $f : X \longrightarrow Y$ are *equivariant* functions $f \in (X \to Y)$. $\diamond$

Definition 4.4.2 (The category FM-Cpo$_\bot$). We take FM-cppos as the objects of the category; the morphisms $f : X \circ\!\!\longrightarrow Y$ are strict, equivariant functions $f \in (X \to Y)$. $\diamond$

It is in fact the case that working in **FM-Cpo** is equivalent to working in the functor category $\mathbf{Cpo}^{\mathbf{I}}$, where $\mathbf{I}$ is the category of finite sets of $\mathbb{A}$ and injections between them. Elements of **FM-Cpo** correspond to elements of $\mathbf{Cpo}^{\mathbf{I}}$ which preserve pullbacks.

Definition 4.4.3 (LFC-functors). A *locally FM-continuous functor* (known as an *LFC-functor*) F : **FM-Cpo**$_\bot \longrightarrow$ **FM-Cpo**$_\bot$ is specified by the following information: for each FM-cppo D, another FM-cppo $F(D)$; and for each function $f \in D \multimap E$, a function $F(f) \in F(D) \multimap F(E)$ preserving identities and composition such that the following conditions are satisfied. Given $f \sqsubseteq f' \in D \multimap E$ then we must have $F(f) \sqsubseteq F(f') \in F(D) \multimap F(E)$. For any finitely-supported chain $(f_n \mid n < \omega)$ in $D \multimap E$, $F(\bigsqcup_{n<\omega} f_n)$ must equal $\bigsqcup_{n<\omega}(F(f_n))$. Finally for all permutations π then $\pi \cdot F$ must equal F—that is to say, for each function $f \in D \multimap E$ then $\pi \cdot (F(f)) = F(\pi \cdot f)$. $\diamond$

Definition 4.4.4 (Mixed-variance LFC-functors). We specify a mixed-variance LFC-functor F : $\mathbf{FM\text{-}Cpo}_{\perp}^{\mathrm{op}} \times \mathbf{FM\text{-}Cpo}_{\perp} \longrightarrow \mathbf{FM\text{-}Cpo}_{\perp}$ by the following information: for each pair of FM-cppos (D, E), another FM-cppo $F(D, E)$; and for each pair of functions $(f, g) \in (E \multimap D) \times (D \multimap E)$, a function $F(f, g) \in F(D, D) \multimap F(E, E)$ such that: $F(id_D, id_D) = id_{F(D,D)}$; and for additional functions $f' \in E' \multimap E$ and $g' \in E \multimap E'$, then $F(f', g') \circ F(f, g) = F(f \circ f', g' \circ g)$. These actions must also satisfy the following conditions. Given $f \sqsubseteq f' \in E \multimap D$ and $g \sqsubseteq g' \in D \multimap E$ then we require $F(f, g) \sqsubseteq F(f', g') \in F(D, D) \multimap F(E, E)$. For any finitely-supported chain $((f_n, g_n) \mid n < \omega)$ in $(E \multimap D) \times (D \multimap E)$, then $F(\bigsqcup_{n<\omega} f_n, \bigsqcup_{n<\omega} g_n)$ must equal $\bigsqcup_{n<\omega}(F(f_n, g_n))$. Finally, for all permutations π and functions $(f, g) \in (E \multimap D) \times (D \multimap E)$ then we must have $\pi \cdot F(f, g) = F(\pi \cdot f, \pi \cdot g)$. $\diamond$

Notation 4.4.5. Write $\underline{\lambda}x.\ t$ for that function which acts as $\lambda x.\ t$ except that $(\underline{\lambda}x.\ t)(\perp) \stackrel{\mathrm{def}}{=} \perp$. Extend this notation in the obvious way to write $\underline{\lambda}\langle d_1, d_2\rangle.\ t$ for strict functions in $D_1 \otimes D_2 \multimap D$ and $\underline{\lambda}[a]d.\ t$ for strict functions in $[\mathbb{A}]D \multimap D'$. Note that this notation imposes no conditions as to which particular representative in $[\mathbb{A}]D$ is chosen. Recall that we write $\langle d_1, d_2\rangle$ to indicate the construction of a smash pair (such that $\langle d_1, d_2\rangle \stackrel{\mathrm{def}}{=} \perp_{D_1 \otimes D_2}$ when either of $d_1 \in D_1$ and $d_2 \in D_2$ are bottom). $\diamond$

Lemma 4.4.6. *The following operations, here expressed in terms of their actions on objects and morphisms, determine LFC-functors.*

lifting	$(-)_\perp$	$\mathbf{FM\text{-}Cpo}_\perp \longrightarrow \mathbf{FM\text{-}Cpo}_\perp$
	$D \mapsto D_\perp$	$f \mapsto \underline{\lambda}x.\ f(x)$.
atom-abstraction	$[\mathbb{A}](-)$	$\mathbf{FM\text{-}Cpo}_\perp \longrightarrow \mathbf{FM\text{-}Cpo}_\perp$
	$D \mapsto [\mathbb{A}]D$	
	$f \mapsto \lambda d.\ [a]f(d \mathbin{@} a)$	*where* $a \in \mathbb{A} \setminus \mathrm{supp}(d, f)$.
smash product	$(-) \otimes (-)$	$\mathbf{FM\text{-}Cpo}_\perp \times \mathbf{FM\text{-}Cpo}_\perp \longrightarrow \mathbf{FM\text{-}Cpo}_\perp$
	$(D, E) \mapsto D \otimes E$	$(f, g) \mapsto \underline{\lambda}\langle d, e\rangle.\ \langle f(d), g(e)\rangle$.
coalesced sum	$(-) \oplus (-)$	$\mathbf{FM\text{-}Cpo}_\perp \times \mathbf{FM\text{-}Cpo}_\perp \longrightarrow \mathbf{FM\text{-}Cpo}_\perp$
	$(D, E) \mapsto D \oplus E$	
	$(f, g) \mapsto \underline{\lambda}s.\ \begin{cases} f(d) & \text{if } s = \mathrm{in}_1(d); \\ g(e) & \text{if } s = \mathrm{in}_2(e). \end{cases}$	
strict function space	$(-) \multimap (-)$	$\mathbf{FM\text{-}Cpo}_\perp^{\mathrm{op}} \times \mathbf{FM\text{-}Cpo}_\perp \longrightarrow \mathbf{FM\text{-}Cpo}_\perp$
	$(D, E) \mapsto D \multimap E$	$(f, g) \mapsto \lambda h.\ g \circ h \circ f$.

Proof. We provide the only case which differs significantly from classical domain theory, that for atom-abstraction.

▶ *Preservation of identities.* Observe that

$$\begin{array}{rcll} ([\mathbb{A}]id_D)([a]d) & = & [a'](([a]d) \mathbin{@} a') & \text{by definition of } [\mathbb{A}]\text{–, with } a' \in \mathbb{A} \setminus \mathrm{supp}(a, d) \\ & = & [a'](a\ a') \cdot d & \text{by definition of } @ \\ & = & [a]d & \text{by Lemma 4.2.28.} \end{array}$$

▶ *Preservation of composition.* Take $f \in D \multimap D'$ and $g \in D' \multimap E$, so[4]:

$$\begin{array}{rcll} [\mathbb{A}](g \circ f)([a]d) = & & & \\ & = & [a']((g \circ f)(([a]d) \mathbin{@} a')) & \text{by definition of } [\mathbb{A}]\text{–, with } a' \in \mathbb{A} \setminus \mathrm{supp}(a, d, f, g) \\ & = & [a']((g \circ f)((a\ a') \cdot d)) & \text{by definition of } @ \\ & = & [a']((a'\ a') \cdot ((g \circ f)((a\ a') \cdot d))) & (a'\ a') \text{ is the identity} \\ & = & ([\mathbb{A}]g \circ [\mathbb{A}]f)([a]d) & \text{as } a' \notin \mathrm{supp}([a']f((a\ a') \cdot d), g). \end{array}$$

We now check that $[\mathbb{A}]$– is a continuous equivariant operator on finitely-supported chains $(f_n \in D \multimap E \mid n < \omega)$.

[4]Note that when we need to satisfy the condition $a' \in \mathbb{A} \setminus \mathrm{supp}([a]d, f)$ (arising from the action of $[\mathbb{A}]$– on morphisms) at the second proof step, we impose a stronger condition to ensure that a' is fresh for g (and indeed a) as well. This a' then automatically satisfies the same side condition which arises in connection with g. That we are able to do this is another example of the 'some/any' property of choosing fresh atoms which runs throughout FM-domain theory. It is also evident in the *Continuity* and *Equivariance* proof cases.

▶ *Monotonicity.* Take $f \sqsubseteq f' \in D \multimap E$. We wish to know that $[\mathbb{A}]f \sqsubseteq [\mathbb{A}]f'$; that is to say for all $[a]d \in [\mathbb{A}]D$ then $([\mathbb{A}]f)([a]d) \sqsubseteq ([\mathbb{A}]f')([a]d)$. Take any $a' \in \mathbb{A} \setminus \mathrm{supp}([a]d, f, f')$. Then $([\mathbb{A}]f)([a]d) = [a']f((a\ a') \cdot d)$ and $([\mathbb{A}]f')([a]d) = [a']f'((a\ a') \cdot d)$. By Lemma 4.2.25 it now suffices to show $f((a\ a') \cdot d) \sqsubseteq f'((a\ a') \cdot d)$ to get the result. But this follows since $f \sqsubseteq f'$.

▶ *Continuity.* We wish to know that $[\mathbb{A}] \bigsqcup_{n<\omega} f_n = \bigsqcup_{n<\omega} [\mathbb{A}]f_n$. That is to say, for all $[a]d$ then $([\mathbb{A}] \bigsqcup_{n<\omega} f_n)([a]d) = (\bigsqcup_{n<\omega} [\mathbb{A}]f_n)([a]d)$. Taking the left-hand side together with some $a' \in \mathbb{A} \setminus \mathrm{supp}\,(f_n \mid n < \omega) \setminus \mathrm{supp}([a]d)$ we obtain

$$
\begin{array}{rcll}
([\mathbb{A}] \bigsqcup\limits_{n<\omega} f_n)([a]d) & = & [a'](\bigsqcup\limits_{n<\omega} f_n)((a\ a') \cdot d) & \text{by definition of } [\mathbb{A}]\text{–, as} \\
 & & & a' \notin \mathrm{supp}(\bigsqcup_{n<\omega} f_n, [a]d) \\
 & = & [a'] \bigsqcup\limits_{n<\omega} f_n((a\ a') \cdot d) & \text{lubs in } D \multimap E \\
 & = & \bigsqcup\limits_{n<\omega} [a']f_n((a\ a') \cdot d) & \text{by Lemma 4.2.32} \\
 & = & \bigsqcup\limits_{n<\omega} ([\mathbb{A}]f_n)([a]d) & \text{by definition of } [\mathbb{A}]\text{–, as} \\
 & & & \forall n,\ a' \notin \mathrm{supp}(f_n, [a]d) \\
 & = & (\bigsqcup\limits_{n<\omega} [\mathbb{A}]f_n)([a]d) & \text{lubs in } [\mathbb{A}]D \multimap [\mathbb{A}]E.
\end{array}
$$

▶ *Equivariance.* We wish to know that for all permutations π and functions $f \in D \multimap E$, $\pi \cdot [\mathbb{A}]f = [\mathbb{A}]\pi \cdot f$. This holds when $\pi \cdot ([\mathbb{A}]f([\pi^{-1} \cdot a]\pi^{-1} \cdot d)) = [\mathbb{A}](\pi \cdot f)([a]d)$ for any $[a]d \in [\mathbb{A}]D$. Taking any $a' \in \mathbb{A} \setminus \mathrm{supp}([\pi^{-1}(a)]\pi^{-1} \cdot d, f)$ then the definition of $[\mathbb{A}]$– implies that the left-hand side of this is equal to $\pi \cdot [a']f((\pi^{-1}(a)\ a') \cdot \pi^{-1} \cdot d)$. This is equivalent to $[\pi(a')]\pi \cdot (f((\pi^{-1}(a)\ a') \cdot \pi^{-1} \cdot d))$ by virtue of the permutation action on $[\mathbb{A}]D$. Recalling that $\pi \cdot f(\pi^{-1} \cdot x) = (\pi \cdot f)(x)$ then we can simplify this to $[\pi(a')](\pi \cdot f)((a\ \pi(a')) \cdot d)$. Next, Lemma 4.1.12 tells us that $a' \in \mathbb{A} \setminus \mathrm{supp}([\pi^{-1}(a)]\pi^{-1} \cdot d, f)$ holds just when $\pi(a') \in \mathbb{A} \setminus \mathrm{supp}([a]d, \pi \cdot f)$. We are now done since we can calculate that $[\pi(a')](\pi \cdot f)((a\ \pi(a')) \cdot d) = [\mathbb{A}](\pi \cdot f)([a]d)$ (we use $\pi(a')$ to satisfy the side-condition on $[\mathbb{A}]$–). □

4.5 Solution of recursive equations on FM-cppos

Perhaps the most important technique in the application of domain theory to the semantics of programming languages is the solution of recursive domain equations[64]. This technique enables us to construct domains whose elements are the denotations of values of user-defined recursive datatypes. The seminal example is the discovery by Scott of a non-trivial solution to the recursive domain equation $D \cong (D \to D)$ in order to represent terms of the λ-calculus.

In the next chapter we will need to solve recursive equations on FM-cppos. In general, such an equation takes the form $\Phi(D) \cong D$, where Φ is built up from the constructors $(-)_\perp$ (lifting), $[\mathbb{A}](-)$ (atom-abstraction), $(-) \otimes (-)$ (smash product), $(-) \oplus (-)$ (coalesced sum) and $(-) \multimap (-)$ (strict continuous function space) together with occurrences of the variable D. What we would like is for Φ to determine an LFC-functor in itself; however, this may not be the case since D might occur in both covariant and contravariant positions. In order to get around this problem, we follow Pitts[46] and rewrite Φ to an LFC-functor $F(D^-, D^+) : \mathbf{FM\text{-}Cpo}_\perp^{\mathrm{op}} \times \mathbf{FM\text{-}Cpo}_\perp \longrightarrow \mathbf{FM\text{-}Cpo}_\perp$ by replacing all occurrences of D in contravariant position with D^- and all occurrences of D in covariant position with D^+. The resulting F is then indeed functorial and it suffices to find an FM-cppo D equipped with an isomorphism $i : F(D, D) \cong D$.

The question, of course, is which solution to choose if there is more than one: we wish for one which it is *canonical* in some sense. The following notion of *minimal invariants* captures this idea of canonicity.

Definition 4.5.1 (Minimal invariant property). The solution (i, D) to the recursive equation $F(D, D) \cong D$ has the *minimal invariant property* if the least fixed point $\mathrm{fix}(\phi)$ of the strict continuous function $\phi \in (D \multimap D) \multimap (D \multimap D)$ defined by $\phi(f) \stackrel{\mathrm{def}}{=} i \circ F(f, f) \circ i^{-1}$ is the identity on D. ⋄

We now prove the following theorem in full. Whilst the proof is arguably standard, we believe that our presentation is elegant in the sense that it cuts down to the bare minimum required to establish the result.

Theorem 4.5.2. *Let* $F : \mathbf{FM\text{-}Cpo}_{\perp}^{\mathrm{op}} \times \mathbf{FM\text{-}Cpo}_{\perp} \longrightarrow \mathbf{FM\text{-}Cpo}_{\perp}$ *be an LFC-functor. Then there exists a solution* (i, D) *satisfying the minimal invariant property with* $i : F(D,D) \cong D$ *and which is unique up to isomorphism.*

Proof. By virtue of F being an LFC-functor, it is a continuous operator on chains of morphisms and therefore given any embedding-projection pair $(e : D \mathrel{\circ\!\!\longrightarrow} E, p : E \mathrel{\circ\!\!\longrightarrow} D)$ we can form another such pair $(F(p,e) : F(D,D) \mathrel{\circ\!\!\longrightarrow} F(E,E), F(e,p) : F(E,E) \mathrel{\circ\!\!\longrightarrow} F(D,D))$ with $F(p,e) \circ F(e,p) \sqsubseteq id_{F(D,D)}$ and $F(e,p) \circ F(p,e) = id_{F(E,E)}$. Since the one-element FM-cppo $\{\perp\}$ is initial for embeddings (the unique morphism being given by the least element of $\{\perp\} \multimap D$ for each FM-cppo D), we can build a chain of embedding-projection pairs as follows:

$$D_0 \underset{p_0}{\overset{e_0}{\rightleftarrows}} D_1 \underset{p_1}{\overset{e_1}{\rightleftarrows}} D_2 \underset{p_2}{\overset{e_2}{\rightleftarrows}} \cdots \tag{4.4}$$

where $D_0 \stackrel{\text{def}}{=} \{\perp\}$, $D_{n+1} \stackrel{\text{def}}{=} F(D_n, D_n)$, e_0 is the unique element of $\{\perp\} \multimap D$, p_0 is the unique element of $D_1 \multimap \{\perp\}$, $e_{n+1} \stackrel{\text{def}}{=} F(p_n, e_n)$ and $p_{n+1} \stackrel{\text{def}}{=} F(e_n, p_n)$. Now define functions $p_{nm} : D_n \mathrel{\circ\!\!\longrightarrow} D_m$ and $e_{nm} : D_n \mathrel{\circ\!\!\longrightarrow} D_m$ as follows.

$$p_{nm} \stackrel{\text{def}}{=} \begin{cases} id_{D_n} & \text{if } m = n; \\ p_{(n-1)m} \circ p_n & \text{if } m < n. \end{cases} \qquad e_{nm} \stackrel{\text{def}}{=} \begin{cases} id_{D_n} & \text{if } m = n; \\ e_{(n+1)m} \circ e_n & \text{if } m > n. \end{cases}$$

This enables us to form an FM-cppo with underlying set

$$D \stackrel{\text{def}}{=} \left\{ d \in \prod_{n<\omega} D_n \;\middle|\; \begin{array}{l} d \text{ has finite support} \\ \text{and for all } m \leq n < \omega, \pi_m(d) = p_{nm}(\pi_n(d)) \end{array} \right\} \tag{4.5}$$

where the π_n $(n < \omega)$ are the usual projections out of a member of the dependent product (and the ordering together with the permutation action is inherited from Definition 4.2.14). Even though this is a subset of an infinitely-indexed product, it is a legal construction because every element is finitely supported.

Next define a second set of embeddings $e'_n : D_n \mathrel{\circ\!\!\longrightarrow} D$ and their associated projections $p'_n : D \mathrel{\circ\!\!\longrightarrow} D_n$ as follows:

$$\begin{array}{rcl} e'_n(d_n) & \stackrel{\text{def}}{=} & (x_m \mid m < \omega), \text{ given } x_m \stackrel{\text{def}}{=} \begin{cases} p_{nm}(d_n) & \text{if } m \leqslant n \\ e_{nm}(d_n) & \text{if } m > n; \end{cases} \\ p'_n(d) & \stackrel{\text{def}}{=} & \pi_n(d) \end{array}$$

where we write $(x_m \mid m < \omega)$ for the tuple $(x_0, x_1, \ldots)$. A consequence of these definitions is that

$$e'_n(p'_n(d)) \stackrel{\text{def}}{=} (x^n_m \mid m < \omega), \text{ given } x^n_m \stackrel{\text{def}}{=} \begin{cases} p_{nm}(\pi_n(d)) & \text{if } m < n \\ \pi_n(d) & \text{if } m = n \\ e_{nm}(\pi_n(d)) & \text{if } m > n. \end{cases}$$

Recalling that the least upper bound of a chain is unaffected by removing finitely many elements from the start of the chain, calculate as follows.

$$\begin{array}{rcl} (\bigsqcup_{n<\omega} (e'_n \circ p'_n))(d) & = & \bigsqcup_{n<\omega} (e'_n \circ p'_n)(d) = \bigsqcup_{n<\omega} (x^n_m \mid m < \omega) \\ & = & (\bigsqcup_{n<\omega} x^n_m \mid m < \omega) = (\bigsqcup_{m<n<\omega} x^n_m \mid m < \omega) \\ & = & (\bigsqcup_{m<n<\omega} (p_{nm} \circ \pi_n)(d) \mid m < \omega) \\ & = & (\bigsqcup_{m<n<\omega} \pi_m(d) \mid m < \omega) \\ & = & (\pi_m(d) \mid m < \omega) = d. \end{array}$$

We now have the important results that

$$p'_n \circ e'_n \;=\; id_{D_n} \tag{4.6}$$

$$\bigsqcup_{n<\omega} e'_n \circ p'_n \quad = \quad id_D. \tag{4.7}$$

The equality (4.7) is the crux. Define the strict continuous functions $i : F(D,D) \circ\!\longrightarrow D$ and $j : D \circ\!\longrightarrow F(D,D)$ as follows:

$$i = \bigsqcup_{n<\omega} e'_{n+1} \circ F(e'_n, p'_n) \quad \text{and} \quad j = \bigsqcup_{n<\omega} F(p'_n, e'_n) \circ p'_{n+1}.$$

Now using (4.7) we have

$$\begin{aligned} i \circ j &= \bigsqcup_{n<\omega} e'_{n+1} \circ F(e'_n, p'_n) \circ F(p'_n, e'_n) \circ p'_{n+1} \\ &= \bigsqcup_{n<\omega} e'_{n+1} \circ id_{F(D_n,D_n)} \circ p'_{n+1} \\ &= \bigsqcup_{n<\omega} e'_{n+1} \circ p'_{n+1} = id_D. \end{aligned}$$

Composing i and j around the other way and using (4.7) together with the local continuity and functorality of F, we have

$$\begin{aligned} j \circ i &= \bigsqcup_{n<\omega} F(p'_n, e'_n) \circ p'_{n+1} \circ e'_{n+1} \circ F(e'_n, p'_n) \\ &= \bigsqcup_{n<\omega} F(p'_n, e'_n) \circ id_D \circ F(e'_n, p'_n) \\ &= \bigsqcup_{n<\omega} F(p'_n, e'_n) \circ F(e'_n, p'_n) \\ &= \bigsqcup_{n<\omega} F(e'_n \circ p'_n, e'_n \circ p'_n) \\ &= F(\bigsqcup_{n<\omega} e'_n \circ p'_n, \bigsqcup_{n<\omega} e'_n \circ p'_n) \\ &= F(id_D, id_D) = id_{F(D,D)}. \end{aligned}$$

Therefore i is an isomorphism; from now on let us write i^{-1} instead of j. Now note that since $p_{mn} = p_n \circ \cdots \circ p_{m-1}$ and $e_{nm} = e_{m-1} \circ \cdots \circ e_n$, we have $F(p_{mn}, e_{nm}) = F(p_{m-1}, e_{m-1}) \circ \cdots \circ F_(p_n, e_n) = e_m \circ \cdots \circ e_{n+1}$. Similarly, $F(e_{nm}, p_{mn}) = p_{n+1} \circ \cdots p_m$. Furthermore, for $k \leqslant n \leqslant m$ one can show by expansion of function compositions that $p_{mk} \circ e_{nm} = p_k \circ \cdots \circ p_{n-1} = p_{nk}$. Similarly, if $n < k \leqslant m$ or $n \leqslant m < k$ then $p_{mk} \circ e_{nm} = e_{nk}$. Combining all of these facts yields that $e'_m \circ e_{nm} = e'_n$ when $n \leqslant m$. In a similar vein, (4.5) immediately gives us that $p_{mn} \circ p'_m = p'_n$ when $n \leqslant m$. We can use this and the previous results to calculate as follows.

$$\begin{aligned} i \circ F(p'_n, e'_n) &= \bigsqcup_{m<\omega} e'_{m+1} \circ F(e'_m, p'_m) \circ F(p'_n, e'_n) \\ &= \bigsqcup_{n\leqslant m<\omega} e'_{m+1} \circ F(p'_n \circ e'_m, p'_m \circ e'_n) \\ &= \bigsqcup_{n\leqslant m<\omega} e'_{m+1} \circ F(p_{mn}, e_{nm}) \\ &= \bigsqcup_{n\leqslant m<\omega} e'_{m+1} \circ e_m \circ \cdots \circ e_{n+1} \\ &= \bigsqcup_{n\leqslant m<\omega} e'_{m+1} \circ e_{(n+1)(m+1)} \\ &= \bigsqcup_{n\leqslant m<\omega} e'_{n+1} = e'_{n+1}. \end{aligned}$$

Symmetrically, we can also calculate the following.

$$\begin{aligned} F(e'_n, p'_n) \circ i^{-1} &= \bigsqcup_{m<\omega} F(e'_n, p'_n) \circ F(p'_m, e'_m) \circ p'_{m+1} \\ &= \bigsqcup_{n\leqslant m<\omega} F(p'_m \circ e'_n, p'_n \circ e'_m) \circ p'_{m+1} \\ &= \bigsqcup_{n\leqslant m<\omega} F(e_{nm}, p_{mn}) \circ p'_{m+1} \end{aligned}$$

$$\begin{aligned} &= \bigsqcup_{n \leqslant m < \omega} p_{n+1} \circ \cdots \circ p_m \circ p'_{m+1} \\ &= \bigsqcup_{n \leqslant m < \omega} p_{(m+1)(n+1)} \circ p'_{m+1} \\ &= \bigsqcup_{n \leqslant m < \omega} p'_{n+1} = p'_{n+1}. \end{aligned}$$

We are now in a position to see that (D, i) has the minimal invariant property. It suffices to show that for all $n < \omega$ then $e'_n \circ p'_n = \phi^n(\bot_{D \multimap D})$ since we can then use (4.7) to deduce that $id_D = \bigsqcup_{n<\omega} e'_n \circ p'_n = \bigsqcup_{n<\omega} \phi^n(\bot_{D \multimap D}) = \mathrm{fix}(\phi)$ as required. By the functorality of F and the results above,

$$\begin{aligned} e'_{n+1} \circ p'_{n+1} &= i \circ F(p'_n, e'_n) \circ F(e'_n, p'_n) \circ i^{-1} \\ &= i \circ F(e'_n \circ p'_n, e'_n \circ p'_n) \circ i^{-1}. \end{aligned}$$

We can now proceed by induction on n to show that $\phi^n(\bot)$ equals $i \circ F(e'_n \circ p'_n, e'_n \circ p'_n) \circ i^{-1}$. The base case ($n = 0$) follows immediately by observing that for any $d \in D$ then $p'_0(d) = \bot$ and thus $e'_0 \circ p'_0 = \bot_{D \multimap D}$ by strictness of e'_0. For the inductive step, assume $\phi^n(\bot) = i \circ F(e'_n \circ p'_n, e'_n \circ p'_n) \circ i^{-1}$. From above, we know that the right-hand side of this is equal to $e'_{n+1} \circ p'_{n+1}$. Therefore

$$\begin{aligned} \phi^{n+1}(\bot) &= i \circ F(\phi^n(\bot), \phi^n(\bot)) \circ i^{-1} \\ &= i \circ F(e'_{n+1} \circ p'_{n+1}, e'_{n+1} \circ p'_{n+1}) \circ i^{-1} \end{aligned}$$

which gives the desired result. To see that the solution (i, D) is unique up to isomorphism, suppose there is another solution (i', E) with the minimal invariant property and consider the following diagram

$$\begin{array}{ccc} (E \multimap D) \times (D \multimap E) & \xrightarrow{\quad\epsilon\quad} & (E \multimap E) \times (D \multimap D) \\ \Big\downarrow{\scriptstyle m} & & \Big\downarrow{\scriptstyle n} \\ (E \multimap D) \times (D \multimap E) & \xrightarrow{\quad\epsilon\quad} & (E \multimap E) \times (D \multimap D) \end{array}$$

where the maps ϵ, m and n are defined as follows.

$$\begin{aligned} &\epsilon \,:\, (f^-, f^+) \mapsto (f^+ \circ f^-, f^- \circ f^+) \\ &m \,:\, (f^-, f^+) \mapsto (i \circ F(f^+, f^-) \circ i'^{-1}, i' \circ F(f^-, f^+) \circ i^{-1}) \\ &n \,:\, (f^-, f^+) \mapsto (i' \circ F(f^-, f^-) \circ i'^{-1}, i \circ F(f^+, f^+) \circ i^{-1}) \end{aligned}$$

Given these definitions it is easy to see that the diagram commutes. Now recall Plotkin's *uniformity principle* which states that for functions $d \in (D \to D)$, $e \in (E \to E)$ and $f \in D \multimap E$ such that $e \circ f = f \circ d$, then $\mathrm{fix}(e) = f \circ \mathrm{fix}(d)$. Since ϵ is clearly strict whilst m and n are continuous (as the isomorphisms i, i' are also and the functor F is an LFC-functor) then we can apply this principle to the diagram above to deduce that $\epsilon(\mathrm{fix}(m)) = \mathrm{fix}(n)$. Writing (g^-, g^+) for $\mathrm{fix}(m)$, observe that $\epsilon(\mathrm{fix}(m)) = (g^+ \circ g^-, g^- \circ g^+) = \mathrm{fix}(n) = (id_E, id_D)$ since (i, D) and (i', D') both have the minimal invariant property. Moreover since $\mathrm{fix}(m) = m(\mathrm{fix}(m))$ we have that $(g^-, g^+) = (i \circ F(g^+, g^-) \circ i'^{-1}, i' \circ F(g^-, g^+) \circ i^{-1})$, meaning that the following diagram commutes.

$$\begin{array}{ccc} F(D, D) & \xrightarrow{\quad i\quad} & D \\ \Big\downarrow{\scriptstyle F(g^+ \circ g^-, g^- \circ g^+)} & & \Big\downarrow{\scriptstyle (g^+, g^-)} \quad \Big\downarrow{\scriptstyle (g^-, g^+)} \\ F(E, E) & \xrightarrow{\quad i'\quad} & E \end{array}$$

We therefore have a mediating isomorphism between D and E. $\square$

4.6 FM-sets of syntax

So far in this chapter we have considered FM-sets of domain-theoretic objects. However, it is important to remember that *we can also form FM-sets of syntactic entities*. In particular, the expressions of Mini-FreshML form an FM-set $\mathcal{E}xp$: the permutation action is given simply by recursively traversing the structure of the expression in question and applying the permutation to any atoms encountered. With respect to this action, it is easy to see that each expression is finitely supported. Since there are no constructs which bind atoms in Mini-FreshML, then the support $\mathrm{supp}(e)$ of some $e \in \mathcal{E}xp$ is equal to the atoms $\mathrm{atms}(e)$ of e. In a similar manner, we can form an FM-set consisting of the frame stacks of Chapter 3.

In the next chapter, we will see how our denotational semantics does in fact determine equivariant functions on FM-sets of syntax. There, we will use the notation $a \# e$ to indicate that $a \notin \mathrm{supp}(e)$ (and similarly for frame stacks and expressions).

5 Mini-FreshML, denotationally

'A mathematician is a machine for turning coffee into theorems.' —Erdös

In the previous chapter we developed a variety of domain theory which we claimed was good for reasoning about names and name binding. In this chapter we support this claim by using it to give a denotational semantics to Mini-FreshML. Our semantics is *computationally adequate*, meaning that equality of denotation implies observational equivalence. This property will enable us to use the denotational semantics to derive operational equivalence results of the type conjectured in Chapter 3.

The major contributions of this chapter are twofold. Firstly, we demonstrate how a standard monad of continuations may be used[61] to provide a model of dynamic allocation. Secondly, we demonstrate how expressions and values in the Mini-FreshML metalanguage correspond to α-equivalence classes of object language terms which we aim to manipulate.

5.1 An overview

In order to model the computational effect of generating fresh names, our denotational semantics is *monadic* in the sense of Moggi[38]. This means that we distinguish between *values* and *computations*. Values of some type τ correspond to elements of an FM-cppo $[\![\tau]\!]$, whilst computations of type τ correspond to elements of an FM-cppo $\mathsf{T}[\![\tau]\!]$, where T is a standard *monad of continuations*.

Given that we are attempting to model the dynamic allocation of names, it may come as a surprise to the reader that we are not using a traditional dynamic allocation monad[37] which keeps track of the names generated during some computation. The reason we do not have to do this is that the finite support properties exhibited by objects in FM-domain theory enable us to keep the allocated names implicit. This fits well with the operational semantics based on frame-stacks of §3.4. Furthermore, FM-domain theory has so far resisted attempts to construct standard dynamic allocation monads within it, as we shall see in §5.1.1.

Let us consider in more detail how we can denote values, expressions and frame stacks. To each Mini-FreshML type τ let us assign an FM-cppo $[\![\tau]\!]$. (We provide the exact definition of this map in due course.) Then we can define the denotation $[\![\Gamma]\!]$ of a typing context Γ as the dependent smash product

$$[\![\Gamma]\!] \stackrel{\text{def}}{=} \bigotimes_{x \in \mathrm{dom}(\Gamma)} [\![\Gamma(x)]\!].$$

We treat a non-bottom $\rho \in [\![\Gamma]\!]$ as a finitely supported function mapping each $x \in \mathrm{dom}(\Gamma)$ to an element $\rho(x) \neq \bot$ of the FM-cppo $[\![\Gamma(x)]\!]$. Then for values v, frame stacks S and expressions e satisfying $\Gamma \vdash v : \tau$, $\Gamma \vdash_s S : \tau \multimap \_$ and $\Gamma \vdash e : \tau$ respectively we provide finitely supported functions of the following kinds:

$$\begin{array}{rcl} \mathcal{V}[\![\Gamma \vdash v : \tau]\!] & \in & [\![\Gamma]\!] \multimap [\![\tau]\!] \\ \mathcal{S}[\![\Gamma \vdash_s S : \tau \multimap \_]\!] & \in & [\![\Gamma]\!] \multimap [\![\tau]\!]^{\bot} \\ \mathcal{E}[\![\Gamma \vdash e : \tau]\!] & \in & [\![\Gamma]\!] \multimap [\![\tau]\!]^{\bot\bot} \end{array}$$

where we define $D^{\bot} \stackrel{\text{def}}{=} D \multimap 1_{\bot}$ for each FM-cppo D. FM-cppos $[\![\tau]\!]$ are the domains of values of type τ whilst FM-cppos $[\![\tau]\!]^{\bot\bot}$ are the domains of computations of type τ.

Intuitively, a non-bottom element $d \in [\![\tau]\!]$ corresponds to a value of type τ. An element $\sigma \in [\![\tau]\!]^{\bot}$ corresponds to a stack accepting a value of type τ and returning $\top$ for termination, or $\bot$ for divergence. Since the behaviour of expressions is determined by the enclosing frame

stack, the denotation $\epsilon \in [\![\tau]\!]^{\perp\perp}$ of some expression in context is a function that accepts the denotation of a frame stack in context and returns either $\top$ or $\perp$. Thus, the denotations of expressions in context lie in the underlying set of a continuation monad $(D \multimap R) \multimap R$, where the 'result' set R is $1_\perp$.

Although we shall not use them explicitly when giving our semantics[1], we do of course have the following standard monad operations.

- ▶ *Unit.* For $d \in D$, $\mathrm{return}\ d \stackrel{\mathrm{def}}{=} \lambda\sigma \in D^\perp.\ \sigma(d)$.
- ▶ *Kleisli lift.* For $e \in D^{\perp\perp}$, $f \in D \multimap E^{\perp\perp}$ then

$$\mathrm{let}\ x \Leftarrow e\ \mathrm{in}\ f\ x \stackrel{\mathrm{def}}{=} \lambda\epsilon \in E^\perp.\ e(\lambda d \in D.\ f\ d\ \epsilon)$$

such that $(\mathrm{let}\ x \Leftarrow e\ \mathrm{in}\ f\ x) \in E^{\perp\perp}$.

We noted earlier that the continuation monad is sufficient to model the dynamic allocation of names—which, as in the operational semantics, will be modelled by atoms $a \in \mathbb{A}$. This may only be done thanks to the finite support property of FM-cppos, namely given some FM-cppo D and any $d \in D$ then we can always find some 'fresh' atom a which does not occur in the support of d. In particular, given the denotation of some frame stack in context $\sigma \in [\![\tau]\!]^\perp$ then we can always pick some $a \notin \mathrm{supp}(\sigma)$. The 'operational' intuition behind this specific case is that such an a corresponds to some atom not yet allocated during the evaluation process. In a nutshell,

> *the denotation of the frame stack encapsulates, via its support, all of the necessary information about 'generated' names at runtime.*

Writing $[\![\mathtt{name}]\!]$ for the FM-cppo of atoms, constructed as $[\![\mathtt{name}]\!] = \mathbb{A}_\perp$, define an element $\mathrm{new} \in [\![\mathtt{name}]\!]^{\perp\perp}$ such that $\mathrm{new}(\sigma \in [\![\mathtt{name}]\!]^\perp) \stackrel{\mathrm{def}}{=} \sigma(a)$ for any $a \notin \mathrm{supp}(\sigma)$. We may always pick such an a because for any σ, $\mathrm{supp}(\sigma)$ is finite and thus $\mathbb{A} \setminus \mathrm{supp}(\sigma)$ is cofinite. Moreover, given some $\sigma \in [\![\mathtt{name}]\!]^\perp$ then the result $\sigma(a)$ is the same no matter which representative $a \notin \mathrm{supp}(\sigma)$ is chosen. For given any $a' \notin \mathrm{supp}(\sigma)$ not equal to a, then

$$\begin{array}{rcll} \sigma(a) & = & ((a\ a') \cdot \sigma)(a) & \text{since } a \notin \mathrm{supp}(\sigma) \\ & = & (a\ a') \cdot (\sigma((a\ a') \cdot a)) & \text{permutation action on functions} \\ & = & (a\ a') \cdot (\sigma(a')) & \text{by definition of transposition} \\ & = & \sigma(a') & \text{since } \top \text{ and } \perp \text{ have empty support.} \end{array}$$

We have that new is monotone and continuous by virtue of elements of $[\![\mathtt{name}]\!]^\perp$ having these properties. It is strict because the least element of $[\![\mathtt{name}]\!]^\perp$ is the constantly-bottom function.

Having defined the morphism new, we immediately obtain the denotation of the `fresh` expression:

$$\mathcal{E}[\![\Gamma \vdash \mathtt{fresh}\ :\ \mathtt{name}]\!](\rho) \stackrel{\mathrm{def}}{=} \mathrm{new}.$$

5.1.1 Dynamic allocation monads

Previous work[63] attempted to make use of a traditional dynamic allocation monad on FM-cppos, but this turned out to have problematic order-theoretic completeness properties. Here, for the record, we briefly review the details of this problem.

Write $\mathbb{P}_{\mathrm{fin}}(\mathbb{A})$ for the FM-set of all finite subsets of $\mathbb{A}$, with permutation action as for Definition 4.1.7. Then for each FM-cppo D let us attempt to construct an FM-cppo $\mathsf{T}(D)$ whose underlying set is the quotient of $D \times \mathbb{P}_{\mathrm{fin}}(\mathbb{A})$ by the equivariant equivalence relation

$$\begin{array}{l} (d, X) \sim (d', X') \stackrel{\mathrm{def}}{\Leftrightarrow} \exists \pi \in \mathit{perm}(\mathbb{A}).\ \mathrm{supp}(d) \setminus X = \mathrm{supp}(d') \setminus X' \wedge \\ \qquad\qquad (\forall a \in \mathrm{supp}(d) \setminus X.\ \pi(a) = a)\ \wedge\ \pi(d) = d' \end{array}$$

and partially-ordered by

$$(d, X) \sqsubseteq (d', X') \stackrel{\mathrm{def}}{\Leftrightarrow} \exists X'' \in \mathbb{P}_{\mathrm{fin}}(\mathbb{A}), e, e'.\ (d, X) \sim (e, X'')\ \wedge\ (d', X') \sim (e', X'')\ \wedge\ e \sqsubseteq e'.$$

[1] Only since when calculating in proofs, we have found it necessary to work with the 'macro-expanded' version of these: our definitions reflect this as we shall see in due course. Simply a 'design decision' as they say in the industry.

$F_{\mathtt{unit}}(-,+)$
$(D,E) \mapsto 1_\bot$
$(f^-,f^+) \mapsto id_{1_\bot}$

$F_{\mathtt{name}}(-,+)$
$(D,E) \mapsto \mathbb{A}_\bot$
$(f^-,f^+) \mapsto id_{\mathbb{A}_\bot}$

$F_\delta(-,+)$
$(D,E) \mapsto E$
$(f^-,f^+) \mapsto id_E$

$F_{\texttt{<<name>>}\tau}(-,+)$
$(D,E) \mapsto [\mathbb{A}]F_\tau(D,E)$
$(f^- \in D' \multimap D, f^+ \in E \multimap E') \mapsto \lambda d \in [\mathbb{A}]F_\tau(D,E).\ [a']F_\tau(f^-,f^+)(d \mathbin{@} a')$
for some/any $a' \in \mathbb{A} \setminus \mathrm{supp}(d,f^-,f^+)$

$F_{\tau\times\tau'}(-,+)$
$(D,E) \mapsto F_\tau(D,E) \otimes F_{\tau'}(D,E)$
$(f^- \in D' \multimap D, f^+ \in E \multimap E') \mapsto$
$\underline{\lambda}\langle d,d'\rangle \in F_\tau(D,E) \otimes F_{\tau'}(D,E).\ \langle F_\tau(f^-,f^+)(d), F_{\tau'}(f^-,f^+)(d')\rangle.$

$F_{\tau\to\tau'}(-,+)$
$(D,E) \mapsto F_\tau(E,D) \multimap (F_{\tau'}(D,E))^{\bot\bot}$
$(f^- \in D' \multimap D, f^+ \in E \multimap E') \mapsto$
$\lambda f \in F_\tau(E,D) \multimap (F_{\tau'}(D,E) \multimap 1_\bot) \multimap 1_\bot.$
$\lambda d \in F_\tau(E',D').\ \lambda f' \in F_{\tau'}(D',E') \multimap 1_\bot.\ f(F_\tau(f^+,f^-)(d))(f' \circ F_{\tau'}(f^-,f^+)).$

Figure 5.1: Actions of functors F_τ : **FM-Cpo**$_\bot^{\mathrm{op}} \times$ **FM-Cpo**$_\bot \longrightarrow$ **FM-Cpo**$_\bot$.

(The permutation action is inherited from the product FM-cppo.) Write $d \setminus X$ for the equivalence class of (d,X) under $\sim$. Elements $d \setminus X$ would correspond to denotations of values d whose computation has involved the generation of the atoms in X.

Pitts has shown[unpublished note] that whilst $\sim$ is indeed an equivariant equivalence relation and $\sqsubseteq$ is likewise an equivariant partial order, $\mathsf{T}(D)$ is unfortunately not an FM-cppo since it does not possess least upper bounds of all finitely-supported chains. To see this, take as D the FM-cppo P from §4.2.4 and let $A_0 \subset A_1 \subset \cdots$ be a strictly increasing chain of finite subsets of atoms. Then the following forms a finitely-supported chain in $\mathsf{T}(D)$:

$$A_0 \setminus A_0 \sqsubseteq A_1 \setminus A_1 \sqsubseteq \cdots .$$

The result of Pitts shows that if $A \setminus A' \sqsubseteq A'' \setminus A'''$ in $\mathsf{T}(D)$ then $|A \cap A'| \leqslant |A'' \cap A'''|$. Therefore if $A \setminus A'$ were to be an upper bound for this chain, then $|A \cap A'| \geqslant |A_n \cap A_n| = |A_n|$ must hold for all n. But this contradicts the finiteness of $A \cap A'$.

Rather than investigate further along this route to determine a correct construction of such a monad T we decided to change tack and use the familiar continuation monad instead. That in itself is arguably far more interesting, as we will see in this chapter.

5.2 Definition of the denotational semantics

5.2.1 Denotation of types

Recall that for each Mini-FreshML type τ then $[\![\tau]\!]$ will be an FM-cppo whose non-$\bot$ elements correspond to closed values of type τ. To construct these, define an LFC-functor F : **FM-Cpo**$_\bot^{\mathrm{op}} \times$ **FM-Cpo**$_\bot \longrightarrow$ **FM-Cpo**$_\bot$:

$$F(-,+) \stackrel{\mathrm{def}}{=} F_{\sigma_1}(-,+) \oplus \cdots \oplus F_{\sigma_K}(-,+)$$

where the family of LFC-functors F_τ is defined by induction on the structure of τ as shown in Figure 5.1. Now we apply Theorem 4.5.2 to deduce the existence of a minimal invariant solution $(i,\mathcal{D})$ to the recursive domain equation $F(D,D) \cong D$. Thus i is an isomorphism from $F(\mathcal{D},\mathcal{D})$ to $\mathcal{D}$ and the identity on $\mathcal{D}$ is $\mathrm{fix}(\phi)$, where ϕ is the function in Definition 4.5.1.

We may now define the denotation $[\![\tau]\!]$ of a type τ as $[\![\tau]\!] \stackrel{\mathrm{def}}{=} F_\tau(\mathcal{D},\mathcal{D})$. It follows that the isomorphism i : $[\![\sigma_1]\!] \oplus \cdots \oplus [\![\sigma_K]\!] \cong [\![\delta]\!]$ serves, along with the in_k, to mediate between $[\![\delta]\!]$ and elements of each $[\![\sigma_k]\!]$.

5.2.2 Denotation of expressions and frame stacks

Each of the maps

$$\begin{array}{rcl} \mathcal{V}[\![\Gamma \vdash v \,:\, \tau]\!] & \in & [\![\Gamma]\!] \multimap [\![\tau]\!] \\ \mathcal{S}[\![\Gamma \vdash_s S \,:\, \tau \multimap \_]\!] & \in & [\![\Gamma]\!] \multimap [\![\tau]\!]^\perp \\ \mathcal{E}[\![\Gamma \vdash e \,:\, \tau]\!] & \in & [\![\Gamma]\!] \multimap [\![\tau]\!]^{\perp\perp} \end{array}$$

sends $\perp$ to itself and is otherwise defined as given in Figures 5.2 and 5.3, using the continuous equivariant function `if` defined as follows:

$$\texttt{if}\ \ a, a', d, d' \stackrel{\text{def}}{=} \begin{cases} d & \text{if } a = a'; \\ d' & \text{otherwise.} \end{cases}$$

The 'continuation-passing style' of the definitions is self-evident. For convenience, when talking about *closed* values, frame stacks and expressions we often abbreviate $\mathcal{V}[\![\emptyset \vdash v \,:\, \tau]\!](\emptyset)$ to $\mathcal{V}[\![v]\!]$, etc. Note that each canonical form v has a denotation *qua* value, $\mathcal{V}[\![\Gamma \vdash v \,:\, \tau]\!] \in [\![\Gamma]\!] \multimap [\![\tau]\!]$, and also *qua* expression, $\mathcal{E}[\![\Gamma \vdash v \,:\, \tau]\!] \in [\![\Gamma]\!] \multimap [\![\tau]\!]^{\perp\perp}$. These two denotations are related by the following lemma.

Lemma 5.2.1. *For a canonical form v such that $\Gamma \vdash v \,:\, \tau$, then $\mathcal{E}[\![\Gamma \vdash v \,:\, \tau]\!] = \mathrm{return} \circ \mathcal{V}[\![\Gamma \vdash v \,:\, \tau]\!]$.*

Proof. By induction on the structure of v.

▶ *Cases (var), (unit), (name) and (fun)* are trivial.

▶ *Case (con).* The induction hypothesis tells us $\mathcal{E}[\![\Gamma \vdash v \,:\, \tau]\!] = \mathrm{return} \circ \mathcal{V}[\![\Gamma \vdash v \,:\, \tau]\!]$. Therefore for $\rho \in [\![\Gamma]\!]$,

$$\begin{array}{ll} & \mathcal{E}[\![\Gamma \vdash \mathtt{C}_k(v) \,:\, \delta]\!](\rho) \\ = & \lambda\sigma \in [\![\delta]\!]^\perp.\, \mathcal{E}[\![\Gamma \vdash v \,:\, \sigma_k]\!](\rho)(\lambda d \in [\![\sigma_k]\!].\, \sigma((i \circ \mathrm{in}_k)(d))) \\ = & \lambda\sigma \in [\![\delta]\!]^\perp.\, \mathrm{return}(\mathcal{V}[\![\Gamma \vdash v \,:\, \sigma_k]\!](\rho))(\lambda d \in [\![\sigma_k]\!].\, \sigma((i \circ \mathrm{in}_k)(d))) \\ = & \lambda\sigma \in [\![\delta]\!]^\perp.\, \sigma((i \circ \mathrm{in}_k)(\mathcal{V}[\![\Gamma \vdash v \,:\, \sigma_k]\!](\rho))) \\ = & (\mathrm{return} \circ \mathcal{V}[\![\Gamma \vdash \mathtt{C}_k(v) \,:\, \delta]\!])(\rho). \end{array}$$

▶ *Case (pair).* The induction hypothesis tells us $\mathcal{E}[\![\Gamma \vdash v \,:\, \tau]\!] = \mathrm{return} \circ \mathcal{V}[\![\Gamma \vdash v \,:\, \tau]\!]$ and $\mathcal{E}[\![\Gamma \vdash v' \,:\, \tau]\!] = \mathrm{return} \circ \mathcal{V}[\![\Gamma \vdash v' \,:\, \tau]\!]$. So for $\rho \in [\![\Gamma]\!]$,

$$\begin{array}{ll} & \mathcal{E}[\![\Gamma \vdash (v,\ v') \,:\, \tau]\!](\rho) \\ = & \lambda\sigma \in [\![\tau \times \tau']\!]^\perp.\, \mathcal{E}[\![\Gamma \vdash v \,:\, \tau]\!](\rho)(\lambda d \in [\![\tau]\!].\, \mathcal{E}[\![\Gamma \vdash v' \,:\, \tau']\!](\rho)(\lambda d' \in [\![\tau']\!].\, \sigma\langle d, d'\rangle)) \\ = & \lambda\sigma \in [\![\tau \times \tau']\!]^\perp. \\ & \quad \mathrm{return}(\mathcal{V}[\![\Gamma \vdash v \,:\, \tau]\!](\rho))(\lambda d \in [\![\tau]\!].\, \mathrm{return}(\mathcal{V}[\![\Gamma \vdash v' \,:\, \tau']\!](\rho))(\lambda d' \in [\![\tau']\!].\, \sigma\langle d, d'\rangle))) \\ = & \lambda\sigma \in [\![\tau \times \tau']\!]^\perp.\, \mathrm{return}(\mathcal{V}[\![\Gamma \vdash v' \,:\, \tau']\!](\rho))(\lambda d' \in [\![\tau']\!].\, \sigma\langle \mathcal{V}[\![\Gamma \vdash v \,:\, \tau]\!](\rho), d'\rangle) \\ = & \lambda\sigma \in [\![\tau \times \tau']\!]^\perp.\, \sigma\langle \mathcal{V}[\![\Gamma \vdash v \,:\, \tau]\!](\rho), \mathcal{V}[\![\Gamma \vdash v' \,:\, \tau']\!](\rho)\rangle \\ = & \mathrm{return}\langle \mathcal{V}[\![\Gamma \vdash v \,:\, \tau]\!](\rho), \mathcal{V}[\![\Gamma \vdash v' \,:\, \tau']\!](\rho)\rangle \\ = & (\mathrm{return} \circ \mathcal{V}[\![\Gamma \vdash (v,\ v') \,:\, \tau \times \tau']\!])(\rho). \end{array}$$

▶ *Case (abst).* As for the pair case: we omit the details. □

Now that we have stated the full definitions of $\mathcal{V}[\![\text{–}]\!]$, $\mathcal{S}[\![\text{–}]\!]$ and $\mathcal{E}[\![\text{–}]\!]$ we are in a position to discuss the denotations of the other non-standard constructs, namely abstraction expressions, expressions which deconstruct abstraction values, and atom-swapping expressions.

We now consider each of these in turn, starting with abstraction expressions. To understand their denotation, we first need to consider the denotation of a canonical form $\texttt{<<}a\texttt{>>}v$, say of type $\texttt{<<name>>}\tau$. (For simplicity, take v to be closed.) This is represented simply as the equivalence class $[a]\mathcal{V}[\![v]\!]$ in $[\mathbb{A}][\![\tau]\!]$. Then the denotation of a non-canonical abstraction expression $\texttt{<<}e\texttt{>>}e'$ such that $\Gamma \vdash \texttt{<<}e\texttt{>>}e' \,:\, \texttt{<<name>>}\tau$ is given by

$$\begin{array}{l} \mathcal{E}[\![\Gamma \vdash \texttt{<<}e\texttt{>>}e' \,:\, \texttt{<<name>>}\tau]\!](\rho) \stackrel{\text{def}}{=} \lambda\sigma \in [\![\texttt{<<name>>}\tau]\!]^\perp. \\ \qquad \mathcal{E}[\![\Gamma \vdash e \,:\, \texttt{name}]\!](\rho)(\underline{\lambda} a \in [\![\texttt{name}]\!].\, \mathcal{E}[\![\Gamma \vdash e' \,:\, \tau]\!](\rho)(\lambda d \in [\![\tau]\!].\, \sigma([a]d))). \end{array}$$

This somewhat complicated-looking clause provides us with a good chance to examine the continuation-passing style of the semantics. Intuitively, the denotation of an abstraction expression acts in such a way as to parallel the following evaluation procedure.

$\mathcal{E}[\![\Gamma \vdash x \,:\, \tau]\!](\rho) \stackrel{\text{def}}{=} \lambda\sigma \in [\![\Gamma(x)]\!]^{\perp}.\ \sigma(\rho(x))$.
$\mathcal{E}[\![\Gamma \vdash \texttt{()} \,:\, \texttt{unit}]\!](\rho) \stackrel{\text{def}}{=} \lambda\sigma \in [\![\texttt{unit}]\!]^{\perp}.\ \sigma(\top)$.
$\mathcal{E}[\![\Gamma \vdash a \,:\, \texttt{name}]\!](\rho) \stackrel{\text{def}}{=} \lambda\sigma \in [\![\texttt{name}]\!]^{\perp}.\ \sigma(a)$.
$\mathcal{E}[\![\Gamma \vdash \mathtt{C}_k\texttt{(}e\texttt{)} \,:\, \delta]\!](\rho) \stackrel{\text{def}}{=} \lambda\sigma \in [\![\delta]\!]^{\perp}.\ \mathcal{E}[\![\Gamma \vdash e \,:\, \sigma_k]\!](\rho)(\lambda d \in [\![\sigma_k]\!].\ \sigma((i \circ \mathrm{in}_k)(d)))$.
$\mathcal{E}[\![\Gamma \vdash \texttt{(}e\texttt{,}\ e'\texttt{)} \,:\, \tau \times \tau']\!](\rho) \stackrel{\text{def}}{=} \lambda\sigma \in [\![\tau \times \tau']\!]^{\perp}.$
$\quad \mathcal{E}[\![\Gamma \vdash e \,:\, \tau]\!](\rho)(\lambda d \in [\![\tau]\!].\ \mathcal{E}[\![\Gamma \vdash e' \,:\, \tau']\!](\rho)(\lambda d' \in [\![\tau']\!].\ \sigma\langle d, d'\rangle))$.
$\mathcal{E}[\![\Gamma \vdash \texttt{fresh} \,:\, \texttt{name}]\!](\rho) \stackrel{\text{def}}{=} \mathrm{new} \stackrel{\text{def}}{=} \lambda\sigma \in [\![\texttt{name}]\!]^{\perp}.\ \sigma(a) \quad (\text{any } a \in \mathbb{A} \setminus \mathrm{supp}(\sigma))$.
$\mathcal{E}[\![\Gamma \vdash \texttt{<<}e\texttt{>>}e' \,:\, \texttt{<<name>>}\tau]\!](\rho) \stackrel{\text{def}}{=} \lambda\sigma \in [\![\texttt{<<name>>}\tau]\!]^{\perp}.$
$\quad \mathcal{E}[\![\Gamma \vdash e \,:\, \texttt{name}]\!](\rho)(\underline{\lambda} a \in [\![\texttt{name}]\!].\ \mathcal{E}[\![\Gamma \vdash e' \,:\, \tau]\!](\rho)(\lambda d \in [\![\tau]\!].\ \sigma([a]d)))$.
$\mathcal{E}[\![\Gamma \vdash \texttt{swap}\ e\texttt{,}\ e'\ \texttt{in}\ e'' \,:\, \tau]\!](\rho) \stackrel{\text{def}}{=} \lambda\sigma \in [\![\tau]\!]^{\perp}.$
$\quad \mathcal{E}[\![\Gamma \vdash e \,:\, \texttt{name}]\!](\rho)(\underline{\lambda} a \in [\![\texttt{name}]\!].$
$\qquad \mathcal{E}[\![\Gamma \vdash e' \,:\, \texttt{name}]\!](\rho)(\underline{\lambda} a' \in [\![\texttt{name}]\!].\ \mathcal{E}[\![\Gamma \vdash e'' \,:\, \tau]\!](\rho)(\lambda d \in [\![\tau]\!].\ \sigma((a\ a') \cdot d))))$.
$\mathcal{E}[\![\Gamma \vdash \texttt{fun}\ f\texttt{(}x\texttt{)}\ \texttt{=}\ e \,:\, \tau \to \tau']\!](\rho) \stackrel{\text{def}}{=}$
$\quad \lambda\sigma \in [\![\tau \to \tau']\!]^{\perp}.\ \sigma(\mathrm{fix}(\underline{\lambda} d \in [\![\tau \to \tau']\!].\ \underline{\lambda} d' \in [\![\tau]\!].$
$\qquad \mathcal{E}[\![\Gamma, f \,:\, \tau \to \tau', x \,:\, \tau \vdash e \,:\, \tau']\!](\rho[f \mapsto d, x \mapsto d'])))$.
$\mathcal{E}[\![\Gamma \vdash e\ e' \,:\, \tau]\!](\rho) \stackrel{\text{def}}{=} \lambda\sigma \in [\![\tau]\!]^{\perp}.$
$\quad \mathcal{E}[\![\Gamma \vdash e \,:\, \tau \to \tau']\!](\rho)(\lambda d \in [\![\tau \to \tau']\!].\ \mathcal{E}[\![\Gamma \vdash e' \,:\, \tau]\!](\rho)(\lambda d' \in [\![\tau]\!].\ (d\ d')(\sigma)))$.
$\mathcal{E}[\![\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e' \,:\, \tau]\!](\rho) \stackrel{\text{def}}{=} \lambda\sigma \in [\![\tau]\!]^{\perp}.$
$\quad \mathcal{E}[\![\Gamma \vdash e \,:\, \tau']\!](\rho)(\underline{\lambda} d' \in [\![\tau']\!].\ \mathcal{E}[\![\Gamma, x \,:\, \tau' \vdash e' \,:\, \tau]\!](\rho[x \mapsto d'])(\sigma))$.
$\mathcal{E}[\![\Gamma \vdash \texttt{let (}x\texttt{,}\ x'\texttt{)}\ \texttt{=}\ e\ \texttt{in}\ e' \,:\, \tau]\!](\rho) \stackrel{\text{def}}{=} \lambda\sigma \in [\![\tau]\!]^{\perp}.$
$\quad \mathcal{E}[\![\Gamma \vdash e \,:\, \tau_1 \times \tau_2]\!](\rho)(\underline{\lambda}\langle d_1, d_2\rangle \in [\![\tau_1 \times \tau_2]\!].$
$\qquad \mathcal{E}[\![\Gamma, x \,:\, \tau_1, x' \,:\, \tau_2 \vdash e' \,:\, \tau]\!](\rho[x \mapsto d_1, x' \mapsto d_2])(\sigma))$.
$\mathcal{E}[\![\Gamma \vdash \texttt{let <<}x\texttt{>>}x' \,\texttt{=}\, e\ \texttt{in}\ e' \,:\, \tau]\!](\rho) \stackrel{\text{def}}{=} \lambda\sigma \in [\![\tau]\!]^{\perp}.$
$\quad \mathcal{E}[\![\Gamma \vdash e \,:\, \texttt{<<name>>}\tau']\!](\rho)(\underline{\lambda}[a]d' \in [\![\texttt{<<name>>}\tau']\!].$
$\qquad \mathcal{E}[\![\Gamma, x \,:\, \texttt{name}, x' \,:\, \tau' \vdash e' \,:\, \tau]\!](\rho[x \mapsto a', x' \mapsto (a\ a') \cdot d'])(\sigma)$
$\qquad (\text{any } a' \in \mathbb{A} \setminus \mathrm{supp}(\sigma, a, d, e, e', \rho))$.
$\mathcal{E}[\![\Gamma \vdash \texttt{if}\ e\ \texttt{=}\ e'\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \,:\, \tau]\!](\rho) \stackrel{\text{def}}{=}$
$\quad \lambda\sigma \in [\![\tau]\!]^{\perp}.\ [\![\Gamma \vdash e \,:\, \texttt{name}]\!](\rho)(\underline{\lambda} a \in [\![\texttt{name}]\!].\ [\![\Gamma \vdash e' \,:\, \texttt{name}]\!](\rho)$
$\qquad (\underline{\lambda} a' \in [\![\texttt{name}]\!].\ \texttt{if}\ a, a', \mathcal{E}[\![\Gamma \vdash e_1 \,:\, \tau]\!](\rho)(\sigma), \mathcal{E}[\![\Gamma \vdash e_2 \,:\, \tau]\!](\rho)(\sigma)))$.
$\mathcal{E}[\![\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ \cdots\ \texttt{|}\ \mathtt{C}_k\texttt{(}x_k\texttt{)}\ \texttt{->}\ e_k\ \texttt{|}\ \cdots \,:\, \tau]\!] \stackrel{\text{def}}{=}$
$\quad \lambda\sigma \in [\![\tau]\!]^{\perp}.\ \mathcal{E}[\![\Gamma \vdash e \,:\, \delta]\!](\rho)(\underline{\lambda} d' \in [\![\delta]\!].$
$\qquad \mathcal{E}[\![\Gamma, x_k \,:\, \sigma_k \vdash e_k \,:\, \tau]\!](\rho[x_k \mapsto d_k])(\sigma))$
$\qquad$ for the unique k and d_k such that $d' = (i \circ \mathrm{in}_k)(d_k)$ when $d' \neq \perp$.

$\mathcal{V}[\![\Gamma \vdash x \,:\, \tau]\!](\rho) \stackrel{\text{def}}{=} \rho(x)$.
$\mathcal{V}[\![\Gamma \vdash \texttt{()} \,:\, \texttt{unit}]\!](\rho) \stackrel{\text{def}}{=} \top$.
$\mathcal{V}[\![\Gamma \vdash a \,:\, \texttt{name}]\!](\rho) \stackrel{\text{def}}{=} a$.
$\mathcal{V}[\![\Gamma \vdash \mathtt{C}_k\texttt{(}v\texttt{)} \,:\, \delta]\!](\rho) \stackrel{\text{def}}{=} (i \circ \mathrm{in}_k)(\mathcal{V}[\![\Gamma \vdash v \,:\, \sigma_k]\!](\rho))$.
$\mathcal{V}[\![\Gamma \vdash \texttt{(}v\texttt{,}\ v'\texttt{)} \,:\, \tau \times \tau']\!](\rho) \stackrel{\text{def}}{=} \langle \mathcal{V}[\![\Gamma \vdash v \,:\, \tau]\!](\rho), \mathcal{V}[\![\Gamma \vdash v' \,:\, \tau']\!](\rho)\rangle$.
$\mathcal{V}[\![\Gamma \vdash \texttt{<<}a\texttt{>>}v \,:\, \texttt{<<name>>}\tau]\!](\rho) \stackrel{\text{def}}{=} [a](\mathcal{V}[\![\Gamma \vdash v \,:\, \tau]\!](\rho))$.
$\mathcal{V}[\![\Gamma \vdash \texttt{fun}\ f\texttt{(}x\texttt{)}\ \texttt{=}\ e \,:\, \tau \to \tau']\!](\rho) \stackrel{\text{def}}{=} \mathrm{fix}(\underline{\lambda} f' \in [\![\tau \to \tau']\!].$
$\quad \underline{\lambda} x' \in [\![\tau]\!].\ \mathcal{E}[\![\Gamma, f \,:\, \tau \to \tau', x \,:\, \tau \vdash e \,:\, \tau']\!](\rho[f \mapsto f', x \mapsto x']))$.

Figure 5.2: Denotation of expressions and canonical forms.

1. Accept a frame stack S expecting an an abstraction value.
2. Evaluate the expression in binding position to yield an atom a.
3. Evaluate the remaining expression to yield a value v.
4. Continue evaluation, having filled the hole in frame stack S with the abstraction value $\texttt{<<}a\texttt{>>}v$.

The sequentiality of this process is encoded in the denotational semantics by identifying the *first* computation to be performed and then passing that a continuation which encodes the *next*

$\mathcal{S}[\![\Gamma \vdash_s [] : \tau \multimap \_]\!](\rho) \stackrel{\text{def}}{=} \lambda x \in [\![\tau]\!].\ \top.$

$\mathcal{S}[\![\Gamma \vdash_s S \circ \mathtt{C}_k(\lceil - \rceil) : \sigma_k \multimap \_]\!](\rho) \stackrel{\text{def}}{=} \lambda v \in [\![\sigma_k]\!].\ \mathcal{S}[\![\Gamma \vdash_s S : \delta \multimap \_]\!](\rho)((i \circ \mathrm{in}_k)(v)).$

$\mathcal{S}[\![\Gamma \vdash_s S \circ (\lceil - \rceil, e) : \tau \multimap \_]\!](\rho) \stackrel{\text{def}}{=} \lambda d \in [\![\tau]\!].$
$\quad \mathcal{E}[\![\Gamma \vdash e : \tau']\!](\rho)(\lambda d' \in [\![\tau']\!].\ \mathcal{S}[\![\Gamma \vdash_s S : \tau \times \tau']\!](\rho)\langle d, d'\rangle).$

$\mathcal{S}[\![\Gamma \vdash_s S \circ (v, \lceil - \rceil) : \tau' \multimap \_]\!](\rho) \stackrel{\text{def}}{=} \lambda d \in [\![\tau']\!].\ \mathcal{S}[\![\Gamma \vdash_s S : \tau \times \tau']\!](\rho)\langle \mathcal{V}[\![\Gamma \vdash v : \tau]\!](\rho), d\rangle.$

$\mathcal{S}[\![\Gamma \vdash_s S \circ \texttt{<<}\lceil - \rceil\texttt{>>}e : \mathtt{name} \multimap \_]\!](\rho) \stackrel{\text{def}}{=} \underline{\lambda} a \in [\![\mathtt{name}]\!].$
$\quad \mathcal{E}[\![\Gamma \vdash e : \tau]\!](\rho)(\lambda d \in [\![\tau]\!].\ \mathcal{S}[\![\Gamma \vdash_s S : \texttt{<<}\mathtt{name}\texttt{>>}\tau]\!](\rho)([a]d)).$

$\mathcal{S}[\![\Gamma \vdash_s S \circ \texttt{<<}v\texttt{>>}\lceil - \rceil : \tau \multimap \_]\!](\rho) \stackrel{\text{def}}{=} \lambda d \in [\![\tau]\!].$
$\quad \mathcal{S}[\![\Gamma \vdash_s S : \texttt{<<}\mathtt{name}\texttt{>>}\tau]\!](\rho)([\mathcal{V}[\![\Gamma \vdash v : \mathtt{name}]\!](\rho)]d).$

$\mathcal{S}[\![\Gamma \vdash_s S \circ \mathtt{swap}\ \lceil - \rceil, e'\ \mathtt{in}\ e'' : \mathtt{name} \multimap \_]\!](\rho) \stackrel{\text{def}}{=} \underline{\lambda} a \in [\![\mathtt{name}]\!].$
$\quad \mathcal{E}[\![\Gamma \vdash e' : \mathtt{name}]\!](\rho)(\underline{\lambda} a' \in [\![\mathtt{name}]\!].\ \mathcal{E}[\![\Gamma \vdash e'' : \tau]\!](\rho)$
$\qquad (\lambda d \in [\![\tau]\!].\ \mathcal{S}[\![\Gamma \vdash_s S : \tau \multimap \_]\!](\rho)((a\ a') \cdot d))).$

$\mathcal{S}[\![\Gamma \vdash_s S \circ \mathtt{swap}\ v, \lceil - \rceil\ \mathtt{in}\ e'' : \mathtt{name} \multimap \_]\!](\rho) \stackrel{\text{def}}{=} \underline{\lambda} a' \in [\![\mathtt{name}]\!].$
$\quad \mathcal{E}[\![\Gamma \vdash e'' : \tau]\!](\rho)(\lambda d \in [\![\tau]\!].\ \mathcal{S}[\![\Gamma \vdash_s S : \tau \multimap \_]\!](\rho)(((\mathcal{V}[\![\Gamma \vdash v : \mathtt{name}]\!](\rho))\ a') \cdot d)).$

$\mathcal{S}[\![\Gamma \vdash_s S \circ \mathtt{swap}\ v, v'\ \mathtt{in}\ \lceil - \rceil : \tau \multimap \_]\!](\rho) \stackrel{\text{def}}{=} \lambda d \in [\![\tau]\!].$
$\quad \mathcal{S}[\![\Gamma \vdash_s S : \tau \multimap \_]\!](\rho)(((\mathcal{V}[\![\Gamma \vdash v : \mathtt{name}]\!](\rho))\ (\mathcal{V}[\![\Gamma \vdash v' : \mathtt{name}]\!](\rho))) \cdot d).$

$\mathcal{S}[\![\Gamma \vdash_s S \circ \lceil - \rceil\ e : (\tau \to \tau') \multimap \_]\!](\rho) \stackrel{\text{def}}{=} \lambda d \in [\![\tau \to \tau']\!].$
$\quad \mathcal{E}[\![\Gamma \vdash e : \tau]\!](\rho)(\lambda d' \in [\![\tau]\!].\ (d\ d')(\mathcal{S}[\![\Gamma \vdash_s S : \tau' \multimap \_]\!](\rho))).$

$\mathcal{S}[\![\Gamma \vdash_s S \circ v\ \lceil - \rceil : \tau \multimap \_]\!](\rho) \stackrel{\text{def}}{=} \lambda d \in [\![\tau]\!].\ ((\mathcal{V}[\![\Gamma \vdash v : \tau \to \tau']\!](\rho))\ d)(\mathcal{S}[\![\Gamma \vdash_s S : \tau' \multimap \_]\!](\rho)).$

$\mathcal{S}[\![\Gamma \vdash_s S \circ \mathtt{let}\ x = \lceil - \rceil\ \mathtt{in}\ e : \tau \multimap \_]\!](\rho) \stackrel{\text{def}}{=}$
$\quad \underline{\lambda} d \in [\![\tau]\!].\ \mathcal{E}[\![\Gamma, x : \tau \vdash e : \tau']\!](\rho[x \mapsto d])(\mathcal{S}[\![\Gamma \vdash_s S : \tau' \multimap \_]\!](\rho)).$

$\mathcal{S}[\![\Gamma \vdash_s S \circ \mathtt{let}\ (x, x') = \lceil - \rceil\ \mathtt{in}\ e : \tau \times \tau' \multimap \_]\!](\rho) \stackrel{\text{def}}{=} \underline{\lambda}\langle d_1, d_2\rangle \in [\![\tau \times \tau']\!].$
$\quad \mathcal{E}[\![\Gamma, x : \tau, x' : \tau' \vdash e : \tau'']\!](\rho[x \mapsto d_1, x' \mapsto d_2])(\mathcal{S}[\![\Gamma \vdash_s S : \tau'' \multimap \_]\!](\rho)).$

$\mathcal{S}[\![\Gamma \vdash_s S \circ \mathtt{let}\ \texttt{<<}x\texttt{>>}x' = \lceil - \rceil\ \mathtt{in}\ e : \texttt{<<}\mathtt{name}\texttt{>>}\tau \multimap \_]\!](\rho) \stackrel{\text{def}}{=}$
$\quad \underline{\lambda}[a]d \in [\![\texttt{<<}\mathtt{name}\texttt{>>}\tau]\!].$
$\qquad \mathcal{E}[\![\Gamma, x : \mathtt{name}, x' : \tau \vdash e : \tau']\!](\rho[x \mapsto a', x' \mapsto (a\ a') \cdot d])$
$\qquad (\mathcal{S}[\![\Gamma \vdash_s S : \tau' \multimap \_]\!](\rho))\quad (a' \in \mathbb{A} \setminus \mathrm{supp}(S, a, d, e, \rho)).$

$\mathcal{S}[\![\Gamma \vdash_s S \circ \mathtt{if}\ \lceil - \rceil = e'\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \tau \multimap \_]\!](\rho) \stackrel{\text{def}}{=} \underline{\lambda} a \in [\![\mathtt{name}]\!].$
$\quad \mathcal{E}[\![\Gamma \vdash e' : \mathtt{name}]\!](\rho)(\underline{\lambda} a' \in [\![\mathtt{name}]\!].$
$\qquad \mathtt{if}\ a, a', \mathcal{E}[\![\Gamma \vdash e_1 : \tau]\!](\rho)(\mathcal{S}[\![\Gamma \vdash_s S : \tau \multimap \_]\!](\rho)),$
$\qquad\quad \mathcal{E}[\![\Gamma \vdash e_2 : \tau]\!](\rho)(\mathcal{S}[\![\Gamma \vdash_s S : \tau \multimap \_]\!](\rho)))$

$\mathcal{S}[\![\Gamma \vdash_s S \circ \mathtt{if}\ v = \lceil - \rceil\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \tau \multimap \_]\!](\rho) \stackrel{\text{def}}{=} \underline{\lambda} a' \in [\![\mathtt{name}]\!].$
$\quad \mathtt{if}\ \mathcal{V}[\![\Gamma \vdash v : \mathtt{name}]\!](\rho), a', \mathcal{E}[\![\Gamma \vdash e_1 : \tau]\!](\rho)(\mathcal{S}[\![\Gamma \vdash_s S : \tau \multimap \_]\!](\rho)),$
$\qquad \mathcal{E}[\![\Gamma \vdash e_2 : \tau]\!](\rho)(\mathcal{S}[\![\Gamma \vdash_s S : \tau \multimap \_]\!](\rho)).$

$\mathcal{S}[\![\Gamma \vdash_s S \circ \mathtt{match}\ \lceil - \rceil\ \mathtt{with}\ \cdots \mid \mathtt{C}_k(x_k) \texttt{->}\ e_k \mid \cdots : \delta \multimap \_]\!] \stackrel{\text{def}}{=}$
$\quad \underline{\lambda} d \in [\![\delta]\!].\ \mathcal{E}[\![\Gamma, x_k : \sigma_k \vdash e_k : \tau]\!](\rho[x_k \mapsto d_k])(\mathcal{S}[\![\Gamma \vdash_s S : \tau \multimap \_]\!](\rho))$
$\quad$ for the unique k and d_k such that $d = (i \circ \mathrm{in}_k)(d_k)$.

Figure 5.3: Denotation of frame stacks.

computation to be performed. In the above definition we are constructing two continuations, which are easily seen to correspond to stages 3 and 4 of the above evaluation procedure:

- $\underline{\lambda} a \in [\![\mathtt{name}]\!].\ \mathcal{E}[\![\Gamma \vdash e' : \tau]\!](\rho)(\lambda d \in [\![\tau]\!].\ \sigma([a]d))$; and
- $\lambda d \in [\![\tau]\!].\ \sigma([a]d)$.

By virtue of their denotations we would expect abstraction values to share the good properties endowed to members of abstraction FM-cppos. A nice example is the algorithm used for testing abstraction values for equality described in §2.2.5. For suppose that we have a closed value $\texttt{<<}a\texttt{>>}v$ as above together with another such value $\texttt{<<}a'\texttt{>>}v'$. Let us say that the denotations of v and v' are d and d' respectively so that the denotations of the two abstraction values are $[a]d$ and $[a']d'$ respectively. Now according to the properties of the partial order $\sqsubseteq$ on abstraction FM-cppos we have that

$$[a]d = [a']d' \Leftrightarrow (a\ a'') \cdot d = (a'\ a'') \cdot d'$$

for some/any fresh $a \in \mathbb{A} \setminus \mathrm{supp}(a, a', d, d')$. This exactly parallels the construction of the equality test[2] for such values in the language. There, we choose a fresh atom a'' and then just compare the values $(a\ a'') \cdot v$ and $(a'\ a'') \cdot v$.

Pattern-matching on abstraction values is provided by the `let <<`x`>>`x' `=` e `in` e' construct. The denotation of such an expression is as follows:

$$\begin{array}{l}\mathcal{E}[\![\Gamma \vdash \texttt{let <<}x\texttt{>>}x' \texttt{ = } e \texttt{ in } e' : \tau]\!](\rho) \stackrel{\mathrm{def}}{=} \lambda\sigma \in [\![\tau]\!]^{\perp}. \\ \quad \mathcal{E}[\![\Gamma \vdash e : \texttt{<<name>>}\tau']\!](\rho)(\underline{\lambda}[a]d' \in [\![\texttt{<<name>>}\tau']\!]. \\ \quad\quad \mathcal{E}[\![\Gamma, x : \texttt{name}, x' : \tau' \vdash e' : \tau]\!](\rho[x \mapsto a', x' \mapsto (a\ a') \cdot d'])(\sigma) \\ \quad (\text{any } a' \in \mathbb{A} \setminus \mathrm{supp}(\sigma, a, d, e, e', \rho)).\end{array}$$

Again whilst this looks complicated, careful inspection will reveal that the construction of the function exactly parallels the procedure for deconstructing abstraction values given in Chapter 2.

Finally, we consider explicit atom-swapping operations. Denotationally, performing such an operation just comes down to exploiting the permutation action of the particular FM-cppo in question as follows.

$$\begin{array}{l}\mathcal{E}[\![\Gamma \vdash \texttt{swap } e, e' \texttt{ in } e'' : \tau]\!](\rho) \stackrel{\mathrm{def}}{=} \lambda\sigma \in [\![\tau]\!]^{\perp}. \\ \quad \mathcal{E}[\![\Gamma \vdash e : \texttt{name}]\!](\rho)(\underline{\lambda}a \in [\![\texttt{name}]\!]. \mathcal{E}[\![\Gamma \vdash e' : \texttt{name}]\!](\rho)(\underline{\lambda}a' \in [\![\texttt{name}]\!]. \\ \quad \mathcal{E}[\![\Gamma \vdash e'' : \tau]\!](\rho)(\lambda d \in [\![\tau]\!]. \sigma((a\ a') \cdot d)))).\end{array}$$

If only implementing swapping in the actual Fresh O'Caml runtime system were so straightforward: we will see in §6.6.2 why it is not.

5.3 Some properties of the semantics

5.3.1 Support and equivariance properties

Recall §4.6 where we introduced the notion of FM-sets of syntax. Here we refine the FM-set $\mathcal{E}xp$ of that section to that of an FM-set of *terms in context*, denoted by $\mathcal{E}$ and defined as follows. The underlying set is

$$\{\, (\Gamma, e, \tau, \rho) \mid \Gamma \vdash e : \tau \ \wedge\ \rho \in [\![\Gamma]\!] \,\}$$

and the permutation action is given by $\pi \cdot (\Gamma, e, \tau, \rho) \stackrel{\mathrm{def}}{=} (\Gamma, \pi \cdot e, \tau, \pi \cdot \rho)$. With respect to this action, each element is indeed finitely supported. The maps $\mathcal{E}[\![\text{–}]\!]$ can now be thought of as elements of the dependent product

$$\prod_{(\Gamma, e, \tau, \rho) \in \mathcal{E}} [\![\tau]\!]^{\perp\perp}.$$

Now we can see that each of these elements is indeed equivariant, since the following lemma tells us that

$$\mathcal{E}[\![\Gamma \vdash \pi \cdot e : \tau]\!](\pi \cdot \rho) = \pi \cdot \mathcal{E}[\![\Gamma \vdash e : \tau]\!](\rho).$$

Similar results hold for $\mathcal{V}[\![\text{–}]\!]$ and $\mathcal{S}[\![\text{–}]\!]$. However it must be noted that the functions denoting values, frame stacks and expressions do not necessarily have empty support by themselves, since atoms may occur throughout values and expressions. As a consequence they are not necessarily morphisms in the category **FM-Cpo**$_{\perp}$ (as one might expect in a categorical semantics, for example). This is easily seen by taking any $a \in \mathbb{A}$ and considering the function $\mathcal{V}[\![\vdash a : \texttt{name}]\!](\emptyset)$, which has a finite support of just $\{a\}$.

Lemma 5.3.1 (Equivariance). *For atoms a, a', values v, frame stacks S and expressions e then*

$$\begin{cases} \Gamma \vdash v : \tau \Rightarrow \forall \rho \in [\![\Gamma]\!].\ \pi \cdot (\mathcal{V}[\![\Gamma \vdash v : \tau]\!](\rho)) = \mathcal{V}[\![\Gamma \vdash \pi \cdot v : \tau]\!](\pi \cdot \rho); \\ \Gamma \vdash_s S : \tau \multimap \_ \Rightarrow \forall \rho \in [\![\Gamma]\!]. \\ \quad \pi \cdot (\mathcal{S}[\![\Gamma \vdash_s S : \tau \multimap \_]\!](\rho)) = \mathcal{S}[\![\Gamma \vdash_s \pi \cdot S : \tau \multimap \_]\!](\pi \cdot \rho); \\ \Gamma \vdash e : \tau \Rightarrow \forall \rho \in [\![\Gamma]\!].\ \pi \cdot (\mathcal{E}[\![\Gamma \vdash e : \tau]\!](\rho)) = \mathcal{E}[\![\Gamma \vdash \pi \cdot e : \tau]\!](\pi \cdot \rho). \end{cases}$$

[2] Of course, in Fresh O'Caml we have more general abstraction types than `<<name>>`τ, but the similarity still holds.

Proof. By induction over the axioms and rules defining the typing relations $\vdash$ and $\vdash_s$. $\square$

Corollary 5.3.2. *For values v, frame stacks S, expressions e and atoms a,*

$$\begin{cases} \Gamma \vdash v \,:\, \tau \Rightarrow \forall \rho \in [\![\Gamma]\!].\ a \mathbin{\#} v, \rho \Rightarrow a \mathbin{\#} \mathcal{V}[\![\Gamma \vdash v \,:\, \tau]\!](\rho); \\ \Gamma \vdash_s S \,:\, \tau \multimap \_ \Rightarrow \forall \rho \in [\![\Gamma]\!].\ a \mathbin{\#} S, \rho \Rightarrow a \mathbin{\#} \mathcal{S}[\![\Gamma \vdash_s S \,:\, \tau \multimap \_]\!](\rho); \\ \Gamma \vdash e \,:\, \tau \Rightarrow \forall \rho \in [\![\Gamma]\!].\ a \mathbin{\#} e, \rho \Rightarrow a \mathbin{\#} \mathcal{E}[\![\Gamma \vdash e \,:\, \tau]\!](\rho). \end{cases}$$

Therefore $\mathrm{supp}(\mathcal{V}[\![\Gamma \vdash v \,:\, \tau]\!](\rho)) \subseteq \mathrm{supp}(v) \cup \mathrm{supp}(\rho)$*; and similarly for frame stacks S and expressions e.*

Proof. The second sentence follows from the first just because $a \mathbin{\#} v$ holds iff $a \notin \mathrm{supp}(v)$ (similarly for S and e). For the first part, recall that $a \mathbin{\#} v$ holds just when $(a\ a') \cdot v = v$ for all $a' \in \mathbb{A} \setminus \mathrm{supp}(v)$. We wish to show that $(a\ a') \cdot (\mathcal{V}[\![\Gamma \vdash v \,:\, \tau]\!](\rho)) = \mathcal{V}[\![\Gamma \vdash v \,:\, \tau]\!](\rho)$, which by the previous Lemma is equivalent to asking $\mathcal{V}[\![\Gamma \vdash (a\ a') \cdot v \,:\, \tau]\!]((a\ a') \cdot \rho) = \mathcal{V}[\![\Gamma \vdash v \,:\, \tau]\!](\rho)$. However, we already have $(a\ a') \cdot v = v$ and $(a\ a') \cdot \rho = \rho$. $\square$

5.3.2 Substitutivity properties

Notation 5.3.3. Write $\mathcal{V}[\![\psi]\!]$ for the element of $[\![\Gamma]\!]$ such that for all $x \in \mathrm{dom}(\Gamma)$, $\mathcal{V}[\![\psi]\!](x) \stackrel{\mathrm{def}}{=} \mathcal{V}[\![\psi(x)]\!]$. $\diamond$

Lemma 5.3.4 (Substitutivity). *For typing contexts Γ, values v, frame stacks S and expressions e then*

$$\begin{cases} \Gamma \vdash v \,:\, \tau \Rightarrow \forall \psi \in \mathrm{Subst}_\Gamma.\ \mathcal{V}[\![e[\psi]]\!] = \mathcal{V}[\![\Gamma \vdash v \,:\, \tau]\!]\mathcal{V}[\![\psi]\!]; \\ \Gamma \vdash_s S \,:\, \tau \Rightarrow \forall \psi \in \mathrm{Subst}_\Gamma.\ \mathcal{S}[\![S[\psi]]\!] = \mathcal{S}[\![\Gamma \vdash_s S \,:\, \tau \multimap \_]\!]\mathcal{V}[\![\psi]\!]; \\ \Gamma \vdash e \,:\, \tau \Rightarrow \forall \psi \in \mathrm{Subst}_\Gamma.\ \mathcal{E}[\![e[\psi]]\!] = \mathcal{E}[\![\Gamma \vdash e \,:\, \tau]\!]\mathcal{V}[\![\psi]\!]. \end{cases}$$

Proof. By induction over the structures of v, S and e. $\square$

5.4 Computational adequacy

In this section we prove a *computational adequacy* result which provides an important bridge between the operational and denotational semantics: it states that expressions with equal denotations are observationally equivalent. The adequacy result is not easy to prove, so at this juncture we simply state the theorem and then postpone its proof until we have developed some further theory.

Theorem 5.4.1 (Adequacy). *Given a typing context Γ and a typeable expression e such that $\Gamma \vdash e \,:\, \tau$, then for all closed frame stacks S of type $\tau \multimap \_$ and substitutions*[3] *$\psi \in \mathrm{Subst}_\Gamma$*

$$\mathcal{E}[\![\Gamma \vdash e \,:\, \tau]\!]\mathcal{V}[\![\psi]\!]\mathcal{S}[\![S]\!] = \top \Leftrightarrow \langle S, e[\psi] \rangle\!\downarrow .$$

Thus if e is a closed expression we have that $\mathcal{E}[\![e]\!]\mathcal{S}[\![S]\!] = \top \Leftrightarrow \langle S, e \rangle\!\downarrow$. $\diamond$

As far as proof is concerned, the forwards direction of the theorem is the difficult case. We adopt the standard technique of using type-indexed *logical relations* which relate domain elements to pieces of syntax.

5.4.1 Construction of the logical relations

We construct three families of relations[4] in the style of [51]:

$$\lhd^{\mathrm{val}}_\tau \subseteq_{\mathrm{eq}} [\![\tau]\!] \times \mathrm{Val}_\tau \qquad \lhd^{\mathrm{stk}}_\tau \subseteq_{\mathrm{eq}} [\![\tau]\!]^\perp \times \mathrm{Stack}_\tau \qquad \lhd^{\mathrm{exp}}_\tau \subseteq_{\mathrm{eq}} [\![\tau]\!]^{\perp\perp} \times \mathrm{Exp}_\tau$$

which must satisfy the following properties. Firstly, for each $v \in \mathrm{Val}_\tau$ then $\left\{ d \mid d \lhd^{\mathrm{val}}_\tau v \right\}$ must contain $\perp$ and be closed under least upper bounds of countable, finitely-supported chains. Furthermore we ask the following:

$$d \lhd^{\mathrm{val}}_{\mathtt{unit}} () \tag{5.1}$$

[3] Recall that ψ maps value identifiers to closed values; thus, $e[\psi]$ must be a closed expression since e is typeable in context Γ.

[4] Note the use of $\subseteq_{\mathrm{eq}}$ to indicate an *equivariant* subset, in the sense of Definition 4.1.6.

$$d \lhd^{\mathrm{val}}_{\mathtt{name}} v \;\Leftrightarrow\; d \neq \bot \Rightarrow d = v \tag{5.2}$$
$$d \lhd^{\mathrm{val}}_{\delta} \mathtt{C}_k(v_k) \;\Leftrightarrow\; \exists d_k \in [\![\sigma_k]\!].\, d = (i \circ \mathrm{in}_k)(d_k) \;\wedge\; d_k \lhd^{\mathrm{val}}_{\sigma_k} v_k \tag{5.3}$$
$$[a_1]d \lhd^{\mathrm{val}}_{\mathtt{<<name>>}\tau} \mathtt{<<}a_2\mathtt{>>}v \;\Leftrightarrow\; (a_1\ a)\cdot d \lhd^{\mathrm{val}}_{\tau} (a_2\ a)\cdot v \text{ for any } a \in \mathbb{A} \setminus \mathrm{supp}(a_1, a_2, d, v) \tag{5.4}$$
$$\langle d_1, d_2\rangle \lhd^{\mathrm{val}}_{\tau\times\tau'} (v_1\mathtt{,}\ v_2) \;\Leftrightarrow\; d_1 \lhd^{\mathrm{val}}_{\tau} v_1 \;\wedge\; d_2 \lhd^{\mathrm{val}}_{\tau'} v_2 \tag{5.5}$$
$$d \lhd^{\mathrm{val}}_{\tau\to\tau'} v \;\Leftrightarrow\; \forall d' \lhd^{\mathrm{val}}_{\tau} v'.\, d(d') \lhd^{\mathrm{exp}}_{\tau'} v\ v' \tag{5.6}$$

The families of auxiliary relations $\lhd^{\mathrm{stk}}_{\tau} \subseteq_{\mathrm{eq}} [\![\tau]\!]^{\bot} \times \mathrm{Stack}_{\tau}$ (relating domain elements σ to frame stacks S of argument type τ) and $\lhd^{\mathrm{exp}}_{\tau} \subseteq_{\mathrm{eq}} [\![\tau]\!]^{\bot\bot} \times \mathrm{Exp}_{\tau}$ (relating domain elements ϵ to expressions e to be evaluated in a frame stack of argument type τ) are defined in terms of $\lhd^{\mathrm{val}}_{\tau}$ as follows.

$$\sigma \lhd^{\mathrm{stk}}_{\tau} S \;\overset{\mathrm{def}}{\Leftrightarrow}\; \forall d \lhd^{\mathrm{val}}_{\tau} v.\, \sigma(d) = \top \Rightarrow \langle S, v\rangle\downarrow \tag{5.7}$$
$$\epsilon \lhd^{\mathrm{exp}}_{\tau} e \;\overset{\mathrm{def}}{\Leftrightarrow}\; \forall \sigma \lhd^{\mathrm{stk}}_{\tau} S.\, \epsilon(\sigma) = \top \Rightarrow \langle S, e\rangle\downarrow \tag{5.8}$$

The meta-notation $\forall d \lhd^{\mathrm{val}}_{\tau} v$ expresses universal quantification over all $d \in [\![\tau]\!]$ and $v \in \mathrm{Val}_{\tau}$ such that $d \lhd^{\mathrm{val}}_{\tau} v$. We read $\forall \sigma \lhd^{\mathrm{stk}}_{\tau} S$ in a similar manner.

The clauses above define the logical relation $\lhd^{\mathrm{val}}_{\tau}$ in terms of a logical relation $\lhd^{\mathrm{val}}_{\delta}$ that is the fixed point of a certain operator acting on relations which we define below. However we cannot simply appeal to Tarski's fixed point theorem to construct this, as the negative occurrence of $\lhd^{\mathrm{val}}_{\tau}$ in clause (5.6) means that this operator is not necessarily monotonic[5]. We thus adapt the 'relational structures' of Pitts[46] in order to proceed.

5.4.2 Relational structures

Definition 5.4.2 (Relational structure). A relational structure $\mathcal{R}$ on the category **FM-Cpo**$_{\bot}$ is specified by the following: for each FM-cppo D, an FM-set $\mathcal{R}(D)$ of '$\mathcal{R}$-relations on D' and an equivariant FM-relation $\subset_{\mathcal{R}} \subseteq (D \multimap E) \times \mathcal{R}(D) \times \mathcal{R}(E)$. We write $f : R \subset S$ iff $f \in D \multimap E$ and $(f, R \in \mathcal{R}(D), S \in \mathcal{R}(E))$ is in $\subset_{\mathcal{R}}$ (abbreviating $\subset_{\mathcal{R}}$ to $\subset$ when the meaning is clear). The relations $\subset$ must satisfy: for every $R \in \mathcal{R}(D)$ then $id_D : R \subset R$ and given a second morphism $g \in E \multimap E'$ with $g : S \subset T$, for $T \in \mathcal{R}(E')$, then $g \circ f : R \subset T$. When $id_D : R \subset R'$ and $id_D : R' \subset R$ imply that $R = R'$ then the relational structure is said to be *normal*. $\diamond$

The particular (normal) relational structure $\mathcal{R}$ which we shall use is defined as follows. For each FM-cppo D, let $\mathcal{R}(D)$ consist of all finitely supported subsets of $D \times \mathrm{Val}_{\tau}$ which contain $\{\bot\} \times \mathrm{Val}_{\tau}$. Write $(d, v : \tau)$ for a member of one of these subsets to make the typing annotation explicit. Since each of these subsets is finitely supported, then $\mathcal{R}(D)$ is indeed an FM-set when equipped with the permutation action $\pi \cdot R \overset{\mathrm{def}}{=} \{(\pi \cdot d, \pi \cdot v : \tau) \mid (d, v : \tau) \in R\}$ for $R \in \mathcal{R}(D)$. For a morphism $f \in D \multimap E$, $R \in \mathcal{R}(D)$ and $S \in \mathcal{R}(E)$ then say that $f : R \subset S$ holds iff $(d, v : \tau) \in R \Rightarrow (f(d), v : \tau) \in S$. The relation $\subset$ has the following important properties.

▷ *Equivariance.* It is easy to see that $\subset$ is supported by the empty set, the permutation action being given by that in Definition 4.1.9.

▷ *Inverse images.* For all $f \in D \multimap E$ and each $S \in \mathcal{R}(E)$ there exists a unique $f^*S \overset{\mathrm{def}}{=} \{(d, v : \tau) \mid (f(d), v : \tau) \in S\}$ such that for all $g \in D \multimap D$ and $R \in \mathcal{R}(D)$, $g : R \subset f^*S \Leftrightarrow f \circ g : R \subset S$.

▷ *Intersections.* Given any finitely-supported subset of relations $\mathcal{S} \subseteq \mathcal{R}(E)$ there exists a unique relation $\bigcap \mathcal{S} \in \mathcal{R}(E)$ (given by set-theoretic intersection of the relations in $\mathcal{S}$) such that for each $g : R \subset \bigcap \mathcal{S}$ then $g : R \subset S$ for each $S \in \mathcal{S}$.

Lemma 5.4.3. *If $i \in E \multimap D$ is an isomorphism then for $R \in \mathcal{R}(D)$ and $S \in \mathcal{R}(E)$, $id_E : S \subset i^*R \Leftrightarrow id_D : (i^{-1})^*S \subset R$.*

Proof. We give the forwards direction, for the reverse is similar. As assumptions we have that $(d, v : \tau) \in S \Rightarrow (i(d), v : \tau) \in R$ and that $(d, v : \tau) \in (i^{-1})^*S$. It follows from the second of these that $(i^{-1}(d), v : \tau) \in S$ by definition of the inverse image operator $(-)^*$. However the first assumption then tells us that $((i \circ i^{-1})(d), v : \tau) \in R$; we conclude since i is iso. $\square$

[5] See [45, 47] for further discussion of such problems.

$$
\begin{array}{rcl}
F_{\mathtt{unit}}(R^-,R^+) & \stackrel{\text{def}}{=} & \{(d, \mathtt{()} \ : \ \mathtt{unit}) \mid d \in 1_\perp\} \\
F_{\mathtt{name}}(R^-,R^+) & \stackrel{\text{def}}{=} & \{(\perp, a \ : \ \mathtt{name}) \mid a \in \mathbb{A}\} \cup \{(a, a \ : \ \mathtt{name}) \mid a \in \mathbb{A}\} \\
F_{\delta}(R^-,R^+) & \stackrel{\text{def}}{=} & R^+ \\
F_{\mathtt{<<name>>}\tau}(R^-,R^+) & \stackrel{\text{def}}{=} & \{([a]d, \mathtt{<<}a'\mathtt{>>}v \ : \ \mathtt{<<name>>}\tau) \mid \\
 & & \exists a'' \in \mathbb{A} \setminus \mathrm{supp}(R^-,R^+,a,a',d,v). \\
 & & ((a\ a'')\cdot d, (a'\ a'')\cdot v \ : \ \tau) \in F_\tau(R^-,R^+)\} \\
F_{\tau\times\tau'}(R^-,R^+) & \stackrel{\text{def}}{=} & \{(\langle d, d'\rangle, \mathtt{(}v\mathtt{,}\ v'\mathtt{)} \ : \ \tau\times\tau') \mid \\
 & & (d, v \ : \ \tau) \in F_\tau(R^-,R^+) \ \wedge \ (d', v' \ : \ \tau') \in F_{\tau'}(R^-,R^+)\} \\
F_{\tau\to\tau'}(R^-,R^+) & \stackrel{\text{def}}{=} & \{(d \in F_\tau^{\mathrm{und}}(R^+,R^-) \multimap (F_{\tau'}^{\mathrm{und}}(R^-,R^+))^{\perp\perp}, \\
 & & \ \mathtt{fun}\ f\mathtt{(}x\mathtt{)}\ \mathtt{=}\ e \ : \ \tau\to\tau') \mid \\
 & & \forall (d',v') \in F_\tau(R^+,R^-), \sigma \in F_{\tau'}^{\mathrm{und}}(R^-,R^+) \multimap 1_\perp, S \ : \ \tau' \multimap \ \_. \\
 & & (\forall (d'',v'') \in F_{\tau'}(R^-,R^+).\ \sigma(d'') = \top \Rightarrow \langle S, v''\rangle\!\downarrow) \Rightarrow \\
 & & (d(d')(\sigma) = \top \Rightarrow \langle S, (\mathtt{fun}\ f\mathtt{(}x\mathtt{)}\ \mathtt{=}\ e)\ v'\rangle\!\Downarrow)\}. \\
 & & \\
F(R^-,R^+) & \stackrel{\text{def}}{=} & \{(\mathrm{in}_k(d_k), \mathtt{C}_k\mathtt{(}v_k\mathtt{)} \ : \ \delta) \mid \\
 & & 1 \leqslant k \leqslant K \ \wedge \ (d_k, v_k \ : \ \sigma_k) \in F_{\sigma_k}(R^-,R^+)\}.
\end{array}
$$

Figure 5.4: Actions on $\mathcal{R}$-relations for F and the F_τ.

Definition 5.4.4 (Admissible relations). Call a relation $R \in \mathcal{R}(D)$ *admissible* iff for each closed canonical form v of type τ, the subset of D given by $\{\, d \in D \mid (d, v \ : \ \tau) \in R\}$ contains $\perp$ and is closed under least upper bounds of countable, finitely-supported chains in D. $\diamond$

Lemma 5.4.5. *The FM-set of admissible relations $R \in \mathcal{R}(D)$, with permutation action inherited from $\mathcal{R}(D)$, forms an FM-complete lattice with greatest lower bounds being given by intersections.*

Proof. Follows from the *Intersections* property above. $\square$

In §5.2.1 we built up FM-cppos $[\![\tau]\!]$ using mixed-variance LFC-functors $\{F, F_\tau\} : \mathbf{FM\text{-}Cpo}_\perp^{\mathrm{op}} \times \mathbf{FM\text{-}Cpo}_\perp \longrightarrow \mathbf{FM\text{-}Cpo}_\perp$. We are now going to equip each such functor with an *equivariant action* on $\mathcal{R}$-relations as shown in Figure 5.4. In that Figure, we use the notation $F_\tau^{\mathrm{und}}(R^-,R^+)$ to stand for the FM-cppo $F_\tau(D^-,D^+)$, where D^- and D^+ are the 'underlying' FM-cppos such that $R^- \in \mathcal{R}(D^-)$ and $R^+ \in \mathcal{R}(D^+)$. The equivariant actions are maps on relations which are order-reversing in their first argument and order-preserving in their second argument, written as $R^-, R^+ \mapsto F(R^-,R^+)$. They build new relations out of old in a way analogous to the way that the functors F and the F_τ construct the FM-cppos used to denote types. We require the actions to satisfy the following properties which we are now going to prove. For $f^- : R_2^- \subset R_1^-$ and $f^+ \ : \ R_1^+ \subset R_2^+$ then

- if R^+ is admissible then $F(R^-,R^+)$ must be also;
- $F(f^-,f^+) \ : \ F(R_1^-,R_1^+) \subset F(R_2^-,R_2^+)$;

and ditto for the F_τ. To proceed, we need the following technical lemma which establishes a now-familiar 'some/any' property.

Lemma 5.4.6. *If $([a]d, \mathtt{<<}a'\mathtt{>>}v \ : \ \mathtt{<<name>>}\tau) \in F_{\mathtt{<<name>>}\tau}(R^-,R^+)$ then for all $a'' \in \mathbb{A} \setminus \mathrm{supp}(R^-,R^+,a,a',d,v)$, $((a\ a'')\cdot d, (a'\ a'')\cdot v \ : \ \tau) \in F_\tau(R^-,R^+)$.*

Proof. Given $([a]d, \mathtt{<<}a'\mathtt{>>}v \ : \ \mathtt{<<name>>}\tau) \in F_{\mathtt{<<name>>}\tau}(R^-,R^+)$ then there exists some $a'' \in \mathbb{A} \setminus \mathrm{supp}(R^-,R^+,a,a',d,v)$ such that $((a\ a'')\cdot d, (a'\ a'')\cdot v \ : \ \tau) \in F_\tau(R^-,R^+)$. Given any other $a''' \in \mathbb{A} \setminus \mathrm{supp}(R^-,R^+,a,a',d,v)$, then $((a''\ a''')\cdot(a\ a'')\cdot d, (a''\ a''')\cdot(a'\ a'')\cdot v \ : \ \tau) \in (a''\ a''')\cdot F_\tau(R^-,R^+)$. Since $a'', a''' \notin \mathrm{supp}(R^-,R^+)$ then this and the equivariance of the F_τ implies $((a''\ a''')\cdot(a\ a'')\cdot d, (a''\ a''')\cdot(a'\ a'')\cdot v \ : \ \tau) \in F_\tau(R^-,R^+)$. Therefore $((a\ a''')\cdot d, (a'\ a''')\cdot v \ : \ \tau) \in F_\tau(R^-,R^+)$ as required. $\square$

Lemma 5.4.7 (Actions on relations preserve admissibility). *For each type τ and $\mathcal{R}$-relations $R^-,\ R^+ \in \mathcal{R}(D)$ then if R^+ is admissible, $F_\tau(R^-, R^+) \in \mathcal{R}(F_\tau(D,D))$ is also (and respectively for the functor F).*

Proof. The case for the functor F is trivial by virtue of the continuity of the injection functions in_k into the smash sum. For the other cases, we proceed by induction on the structure of τ. All of these except the following two are straightforward and we omit them.

► *Case (abst).* We need to show that when R^+ and $F_\tau(R^-, R^+)$ satisfy the admissibility property then so does $F_{\texttt{<<name>>}\tau}(R^-, R^+)$, i.e. that the FM-set

$$\left\{\, [a]d \mid ([a]d, \texttt{<<}a'\texttt{>>}v' \ :\ \texttt{<<name>>}\tau) \in F_{\texttt{<<name>>}\tau}(R^-, R^+)\right\}$$

partially ordered as for $F_{\texttt{<<name>>}\tau}(D,D)$ contains $[a]\bot$ (for any $a \in \mathbb{A}$) and has least upper bounds of finitely-supported chains $([a_n]d_n \mid n < \omega) \in F_{\texttt{<<name>>}\tau}(D,D)$ for some fixed $\texttt{<<}a'\texttt{>>}v'$. It is easy to see that it contains the aforementioned least element. Moreover, Lemma 4.2.32 tells us that for any $a \notin \mathrm{supp}\,([a_n]d_n \mid n < \omega)$ such a finitely-supported chain has a least upper bound $[a]\bigsqcup_{n<\omega}(([a_n]d_n)@a)$, which is equal to $[a]\bigsqcup_{n<\omega}(a\ a_n)\cdot d_n$; we can calculate that this lies in the FM-set $\{\, d \mid (d, v\ :\ \tau) \in F_\tau(R^-, R^+)\}$ by virtue of the induction hypothesis.

► *Case (fn).* This case is straightforward, save for the crucial observation that for a chain $((d_n, \texttt{fun}\ f\texttt{(}x\texttt{)}\ \texttt{=}\ e\ :\ \tau \to \tau') \mid n < \omega) \in F_{\tau\to\tau'}(R^-, R^+)$ and d', σ as in Figure 5.4, then the fact that $(\bigsqcup_{n<\omega} d_n)(d')(\sigma) = \bigsqcup_{n<\omega}(d_n(d')(\sigma)) = \top$ implies there exists some n such that $d_n(d')(\sigma) = \top$. □

Lemma 5.4.8 (Actions on relations preserve $\subset$). *For each Mini-FreshML type τ, $\mathcal{R}$-relations $R_1^- \in \mathcal{R}(D_1^-)$, $R_2^- \in \mathcal{R}(D_2^-)$, $R_1^+ \in \mathcal{R}(D_1^+)$, $R_2^+ \in \mathcal{R}(D_2^+)$ and elements $f^- \in D_2^- \multimap D_1^-$, $f^+ \in D_1^+ \multimap D_2^+$ such that $f^-\ :\ R_2^- \subset R_1^-$ and $f^+\ :\ R_1^+ \subset R_2^+$ then*

$$(d, v\ :\ \tau) \in F_\tau(R_1^-, R_1^+) \Rightarrow (F_\tau(f^-, f^+)(d), v\ :\ \tau) \in F_\tau(R_2^-, R_2^+)$$

and similarly for the functor F.

Proof. For the first part we proceed by induction on the structure of τ. When considering elements $(d, v\ :\ \tau)$ of each $F_\tau(R_1^-, R_1^+)$ in turn, we omit the $d = \bot$ cases since these go through immediately (by virtue of the admissibility of $F_\tau(R_2^-, R_2^+)$).

► *Cases (unit)–(data)* are straightforward, by virtue of $F_{\texttt{unit}}$, $F_{\texttt{name}}$ and F_δ being constant functors.

► *Case (abst).* Take $([a]d, \texttt{<<}a'\texttt{>>}v\ :\ \texttt{<<name>>}\tau) \in F_{\texttt{<<name>>}\tau}(R_1^-, R_1^+)$. For any $a'' \in \mathbb{A} \setminus \mathrm{supp}(R_1^-, R_1^+, R_2^-, R_2^+, a, d, a', v')$, Lemma 5.4.6 tells us that $((a\ a'')\cdot d, (a'\ a'')\cdot v\ :\ \tau) \in F_\tau(R_1^-, R_1^+)$. The induction hypothesis then implies that $(F_\tau(f^-, f^+)((a\ a'')\cdot d), (a'\ a'')\cdot v : \tau) \in F_\tau(R_2^-, R_2^+)$ and moreover by equivariance of F_τ, $((a\ a'')\cdot F_\tau(f^-, f^+)(d), (a'\ a'')\cdot v\ :\ \tau) \in F_\tau(R_2^-, R_2^+)$. Since $F_\tau(f^-, f^+)$ has empty support, this implies $([a](F_\tau(f^-, f^+)(d)), \texttt{<<}a'\texttt{>>}v\ :\ \tau) \in F_{\texttt{<<name>>}\tau}(R_2^-, R_2^+)$ and thus, $(F_{\texttt{<<name>>}\tau}(f^-, f^+)([a]d), \texttt{<<}a'\texttt{>>}v : \tau) \in F_{\texttt{<<name>>}\tau}(R_2^-, R_2^+)$.

► *Case (pair).* Consider an element $(d, v\ :\ \tau \times \tau')$ of $F_{\tau\times\tau'}(R_1^-, R_1^+)$. By the induction hypothesis we have that for all $(d_1, v_1\ :\ \tau) \in F_\tau(R_1^-, R_1^+)$ and $(d_2, v_2\ :\ \tau') \in F_{\tau'}(R_1^-, R_1^+)$ then

$$(F_\tau(f^-, f^+)(d_1), v_1\ :\ \tau) \in F_\tau(R_2^-, R_2^+) \text{ and} \tag{5.9}$$
$$(F_{\tau'}(f^-, f^+)(d_2), v_2\ :\ \tau') \in F_{\tau'}(R_2^-, R_2^+). \tag{5.10}$$

Writing d as $\langle d_1', d_2'\rangle$, v as $(v_1,\ v_2)$ and observing the action of the functor $F_{\tau_1\times\tau_2}$ in Figure 5.4 we see that $(d_1', v_1\ :\ \tau) \in F_\tau(R^-, R^+)$ and $(d_2', v_2\ :\ \tau') \in F_{\tau'}(R^-, R^+)$; thus, we can apply (5.9) and (5.10) to deduce that $(F_\tau(f^-, f^+)(d_1'), v_1) \in F_\tau(R_2^-, R_2^+) \wedge (F_{\tau'}(f^-, f^+)(d_2'), v_2) \in F_{\tau'}(R_2^-, R_2^+)$. We then conclude by observing the action of $F_{\tau\times\tau'}$ given in Figure 5.4.

► *Case (fn).* Consider an element $(d, \texttt{fun}\ f\texttt{(}x\texttt{)}\ \texttt{=}\ e)$ of $F_{\tau\to\tau'}(R_1^-, R_1^+)$. Thus we have that for all $(d_1^\tau, v_1^\tau) \in F_\tau(R_1^+, R_1^-)$, $\sigma \in F_{\tau'}(D_1^-, D_1^+) \multimap 1_\bot$ and $S\ :\ \tau' \multimap \_$

$$\begin{array}{l}(\forall (d_1^{\tau'}, v_1^{\tau'}) \in F_{\tau'}(R_1^-, R_1^+).\ \sigma(d_1^{\tau'}) = \top \Rightarrow \langle S, v_1^{\tau'}\rangle\downarrow) \Rightarrow \\ \quad (d(d_1^\tau)(\sigma) = \top \Rightarrow \langle S, (\texttt{fun}\ f\texttt{(}x\texttt{)}\ \texttt{=}\ e)\ v'\rangle\Downarrow)\}.\end{array} \tag{5.11}$$

The actions of the F_τ and $F_{\tau'}$ provide maps $F_\tau(f^+, f^-) \in F_\tau(D_2^+, D_2^-) \multimap F_\tau(D_1^+, D_1^-)$ and $F_{\tau'}(f^-, f^+) \in F_{\tau'}(D_1^-, D_1^+) \multimap F_{\tau'}(D_2^-, D_2^+)$. Thus by the induction hypothesis we have that

$$(d_2^\tau, v_2^\tau\ :\ \tau) \in F_\tau(R_2^+, R_2^-) \Rightarrow (F_\tau(f^+, f^-)(d_2^\tau), v_2^\tau\ :\ \tau) \in F_\tau(R_1^+, R_1^-), \tag{5.12}$$

$$(d_1^{\tau'}, v_1^{\tau'} \,:\, \tau') \in F_{\tau'}(R_1^-, R_1^+) \Rightarrow (F_{\tau'}(f^-, f^+)(d_1^{\tau'}), v_1^{\tau'} \,:\, \tau') \in F_{\tau'}(R_2^-, R_2^+). \quad (5.13)$$

We need for all such (d_2^τ, v_2^τ), $\sigma' \in F_{\tau'}(D_2^-, D_2^+) \multimap 1_\perp$ and $S \,:\, \tau' \multimap \_$,

$$F_{\tau\to\tau'}(f^-, f^+)(d)(d_2^\tau)(\sigma') = \top \Rightarrow \langle S, (\texttt{fun}\ \ f(x)\ \texttt{=}\ e)\ v_2^\tau\rangle\downarrow \quad (5.14)$$

under the assumption that:

$$\forall (d_2^{\tau'}, v_2^{\tau'}) \in F_{\tau'}(R_2^-, R_2^+).\ \sigma'(d_2^{\tau'}) = \top \Rightarrow \langle S, v_2^{\tau'}\rangle\downarrow . \quad (5.15)$$

We can unpack the functorial action[6] in (5.14) to see that we must prove

$$d(F_\tau(f^+, f^-)(d_2^\tau))(\sigma' \circ F_{\tau'}(f^-, f^+)) = \top \Rightarrow \langle S, (\texttt{fun}\ \ f(x)\ \texttt{=}\ e)\ v_2^\tau\rangle\downarrow . \quad (5.16)$$

Given any $(d_1^{\tau'}, v_1^{\tau'}) \in F_{\tau'}(R_1^-, R_1^+)$ then we can combine (5.13) with (5.15) to deduce $(\sigma' \circ F_{\tau'}(f^-, f^+))(d_1^{\tau'}) = \top \Rightarrow \langle S, v_1^{\tau'}\rangle\downarrow$. We do a sanity check that $\sigma' \circ F_{\tau'}(f^-, f^+)$ does indeed map from $F_{\tau'}(D_1^+, D_1^-)$ to $1_\perp$. Then from (5.11) we have that for all $(d_1^\tau, v_1^\tau) \in F_\tau(R_1^+, R_1^-)$,

$$d(d_1^\tau)(\sigma' \circ F_{\tau'}(f^-, f^+)) = \top \Rightarrow \langle S, (\texttt{fun}\ \ f(x)\ \texttt{=}\ e)\ v_1^\tau\rangle\downarrow . \quad (5.17)$$

Now, (5.12) tells us that $(F_\tau(f^+, f^-)(d_2^\tau), v_2^\tau)$ lies in $F_\tau(R_1^+, R_1^-)$. Using this pair as the (d_1^τ, v_1^τ) in (5.17) then yields (5.16) as required.

Considering now the functor F, take an element $(d, \mathtt{C}_k(v_k) : \delta) \in F(R_1^-, R_1^+)$ such that $1 \leqslant k \leqslant K$. We may write any such d as $\mathrm{in}_k(d_k)$. Then we have that $(d_k, v_k \,:\, \sigma_k) \in F_{\sigma_k}(R_1^-, R_1^+)$. We need to show that $(F(f^-, f^+)(\mathrm{in}_k(d_k)), \mathtt{C}_k(v_k) \,:\, \delta) \in F(R_2^-, R_2^+)$. Using the earlier parts of this lemma and the assumption $(d_k, v_k \,:\, \sigma_k) \in F_{\sigma_k}(R_1^-, R_1^+)$ we now deduce

$$(F_{\sigma_k}(f^-, f^+)(d_k), v_k \,:\, \sigma_k) \in F_{\sigma_k}(R_2^-, R_2^+)$$

Thus $(\mathrm{in}_k(F_{\sigma_k}(f^-, f^+)(d_k)), \mathtt{C}_k(v_k) \,:\, \delta)$ lies in $F_\delta(R_2^-, R_2^+)$. But by the action of the functor F this is just $(F(f^-, f^+)(\mathrm{in}_k(d_k)), \mathtt{C}_k(v_k) \,:\, \delta)$. □

What we are ultimately interested in showing is that a certain *invariant relation* exists as stated in the following Lemma, whose proof we defer for just a moment.

Lemma 5.4.9 (Existence of invariant relations). *There exists an invariant relation Δ satisfying the following properties:*

$$i \,:\, F(\Delta, \Delta) \subset \Delta \quad \textit{and} \quad i^{-1} \,:\, \Delta \subset F(\Delta, \Delta),$$

where i is the isomorphism part of the minimal invariant solution to the recursive equation on FM-cppos given in §5.2.1. ◇

As we shall see, the very fact that the solution to the recursive equation on FM-cppos satisfies the minimal invariant property is crucial to deducing the existence of Δ. Now it turns out that the relation Δ is in fact the crux of the matter, because we will define

$$\lhd_\tau^{\mathrm{val}} \stackrel{\mathrm{def}}{=} F_\tau(\Delta, \Delta)$$

for each type τ and the game is over. Let us now give the proof.

Proof. Observe that

$$\begin{array}{lll} & i^{-1} \,:\, \Delta \subset F(\Delta, \Delta) & \\ \Leftrightarrow & i^{-1} \circ id_D \,:\, \Delta \subset F(\Delta, \Delta) & \\ \Leftrightarrow & id_D \,:\, \Delta \subset (i^{-1})^* F(\Delta, \Delta) & \text{by inverse images} \\ \Leftrightarrow & \Delta \subseteq (i^{-1})^* F(\Delta, \Delta) & \text{by definition of } - \subset - \end{array}$$

and moreover

$$\begin{array}{ll} & i \,:\, F(\Delta, \Delta) \subset \Delta \\ \Leftrightarrow & i \circ id_{F(D,D)} \,:\, F(\Delta, \Delta) \subset \Delta \end{array}$$

[6] See Figure 5.1 for the definition.

$$\begin{array}{lll} \Leftrightarrow & id_{F(D,D)} \,:\, F(\Delta,\Delta) \subset i^*\Delta & \text{by inverse images} \\ \Leftrightarrow & id_D \,:\, (i^{-1})^* F(\Delta,\Delta) \subset \Delta & \text{by Lemma 5.4.3} \\ \Leftrightarrow & (i^{-1})^* F(\Delta,\Delta) \subseteq \Delta & \text{by definition of } - \subset -. \end{array}$$

Thus in order to deduce the existence of the invariant relation Δ it suffices to obtain a fixed point solution to the equation $\Delta = \Phi(\Delta)$, where $\Phi(R) \stackrel{\text{def}}{=} (i^{-1})^* F(R,R)$. We can see that Φ is an equivariant operator on the FM-complete lattice $\mathcal{L}$ of admissible relations $R \in \mathcal{R}(D)$. But whilst Φ is equivariant, it is not necessarily monotonic due to the possible mixed-variance of F. Thus, we define a second operator

$$\Psi(R^-, R^+) \stackrel{\text{def}}{=} (i^{-1})^* F(R^-, R^+)$$

which we claim is a monotone, equivariant function from the FM-complete lattice $\mathcal{L}^{\text{op}} \times \mathcal{L}$ to $\mathcal{L}$. Let us check its monotonicity, for the equivariance is easy to see. Given admissible relations $R_1^-, R_2^-, R_1^+, R_2^+$ in $\mathcal{L}$ such that $id_D \,:\, R_2^- \subset R_1^-$ and $id_D \,:\, R_1^+ \subset R_2^+$, then we have that

$$F(id_D, id_D) \,:\, F(R_1^-, R_1^+) \subset F(R_2^-, R_2^+)$$

by virtue of the admissible actions of the functor F. Since $F(id_D, id_D) = id_{F(D,D)}$ then this yields that $F(R_1^-, R_1^+) \subseteq F(R_2^-, R_2^+)$; the monotonicity of Ψ then arises since the inverse image operator $(i^{-1})^*$ is monotone. It follows that the operator

$$\Psi'(R^-, R^+) \stackrel{\text{def}}{=} (\Psi(R^+, R^-), \Psi(R^-, R^+))$$

is a monotone, equivariant function from the FM-complete lattice $\mathcal{L}^{\text{op}} \times \mathcal{L}$ to itself. This now 'type-checks' correctly in order that we can appeal to Tarski (Lemma 4.3.4) to obtain the least fixed point of Ψ'. Let us denote this by (Δ^-, Δ^+). Since Ψ' is an equivariant function then we can apply Lemma 4.5 to deduce that Δ^- and Δ^+ possess empty support. Observe further that if we can prove that $\Delta^- = \Delta^+ = \Delta$ then

$$(\Delta,\Delta) = \Psi'(\Delta,\Delta) = (\Psi(\Delta,\Delta), \Psi(\Delta,\Delta)) = (\Phi(\Delta), \Phi(\Delta))$$

and thus $\Delta = \Phi(\Delta)$ as we desire. Given that (Δ^-, Δ^+) is a fixed point of Ψ', we calculate that

$$\Delta^- = \Psi(\Delta^+, \Delta^-) \quad \text{and} \quad \Delta^+ = \Psi(\Delta^-, \Delta^+); \tag{5.18}$$

moreover, since it is the *least* fixed point we also have that for any admissible relations $(R^-, R^+) \in \mathcal{L}^{\text{op}} \times \mathcal{L}$,

$$\begin{array}{l} (id_D \,:\, R^- \subset \Psi(R^+, R^-) \;\wedge\; id_D \,:\, \Psi(R^-, R^+) \subset R^+) \Rightarrow \\ \quad id_D \,:\, R^- \subset \Delta^- \;\wedge\; id_D \,:\, \Delta^+ \subset R^+. \end{array} \tag{5.19}$$

To prove that $\Delta^- = \Delta^+$ then we just need to show that $id_D \,:\, \Delta^+ \subset \Delta^-$ and $id_D \,:\, \Delta^- \subset \Delta^+$. We have the first of these straight away, for setting $R^- = \Delta^+$ and $R^+ = \Delta^-$ in (5.19) yields

$$(id_D \,:\, \Delta^+ \subset \Psi(\Delta^-, \Delta^+) \;\wedge\; id_D \,:\, \Psi(\Delta^+, \Delta^-) \subset \Delta^-) \Rightarrow id_D \,:\, \Delta^+ \subset \Delta^-,$$

the left-hand side of which holds by virtue of (5.18).

The reverse inclusion ($id_D \,:\, \Delta^- \subset \Delta^+$) is the more interesting case as its proof is critically dependent on the minimal invariance property of the FM-cppo D, namely that the identity on D is the least fixed point of the function $\phi(f) \stackrel{\text{def}}{=} i \circ F(f,f) \circ i^{-1}$. For consider the FM-cppo of functions

$$S \stackrel{\text{def}}{=} \left\{ f \in D \multimap D \mid f \,:\, \Delta^- \subset \Delta^+ \right\},$$

which contains the least element in $D \multimap D$ (by admissibility of Δ^+) and has least upper bounds for all finitely supported chains. Observe that it suffices to prove that $f \in S \Rightarrow \phi(f) \in S$, for then we can appeal to Scott induction to get $id_D = \text{fix}(\phi) \in S$ which yields $id_D \,:\, \Delta^- \subset \Delta^+$ as desired. Now given such an f we can calculate that $F(f,f) : F(\Delta^+, \Delta^-) \subset F(\Delta^-, \Delta^+)$. Secondly, by virtue of inverse images and Lemma 5.4.3 once again it is easy to see that the equations (5.18) imply $i^{-1} \,:\, \Delta^- \subset F(\Delta^+, \Delta^-)$ and $i \,:\, F(\Delta^-, \Delta^+) \subset \Delta^+$. Thus by transitivity of $- \subset -$, $i \circ F(f,f) \circ i^{-1} \,:\, \Delta^- \subset \Delta^+$, showing that $\phi(f)$ does indeed lie in S. Therefore $\Delta^- = \Delta^+ = \Delta$, meaning that $i \,:\, F(\Delta,\Delta) \subset \Delta$ and $i^{-1} \,:\, \Delta \subset F(\Delta,\Delta)$. $\square$

Since $\lhd_\tau^{\mathrm{stk}}$ and $\lhd_\tau^{\mathrm{exp}}$ are defined in terms of $\lhd_\tau^{\mathrm{val}}$, then the following theorem suffices (at last!) to establish the existence of the logical relations.

Theorem 5.4.10 (Existence of each $\lhd_\tau^{\mathrm{val}}$ relation). *Each logical relation for values may be constructed as* $\lhd_\tau^{\mathrm{val}} \stackrel{\mathrm{def}}{=} F_\tau(\Delta, \Delta)$.

Proof. Since $\Delta \in \mathcal{R}(D)$, then $F_\tau(\Delta, \Delta)$ lies in $\mathcal{R}(F_\tau(D, D))$, which is in turn equal to $\mathcal{R}[\![\tau]\!]$. Thus, $\lhd_\tau^{\mathrm{val}} \subseteq F_\tau(\Delta, \Delta) \subseteq [\![\tau]\!] \times \mathrm{Val}_\tau$. Moreover, since Δ and F_τ both have empty support then we indeed have that $\lhd_\tau^{\mathrm{val}} \subseteq_{\mathrm{eq}} [\![\tau]\!] \times \mathrm{Val}_\tau$. By virtue of the admissibility properties of Δ, then for some typeable canonical form v of type τ we have that $\{d \mid d \lhd_\tau^{\mathrm{val}} v\}$ contains $\bot$ and is closed under least upper bounds of finitely-supported chains. Properties (5.1), (5.2), (5.4), (5.5) and (5.6) follow immediately from the definitions of $F_{\mathtt{unit}}$, $F_{\mathtt{name}}$, $F_{\mathtt{<<name>>}\tau}$, $F_{\tau_1 \times \tau_2}$ and $F_{\tau_1 \to \tau_2}$ respectively, together with Lemma 5.4.6. Property (5.3) follows by first observing that $\lhd_\delta^{\mathrm{val}} = F_\delta(\Delta, \Delta) = \Delta$; then, if $(d, v) \in \Delta$ we have $(i^{-1}(d), v) \in F(\Delta, \Delta)$ (since $i^{-1} : \Delta \subset F(\Delta, \Delta)$). Therefore v is of the form $\mathtt{C}_k\mathtt{(}v_k\mathtt{)}$ and there exists d_k with $i^{-1}(d) = \mathrm{in}_k(d_k)$. It follows that $d = (i \circ \mathrm{in}_k)(d_k)$ and $(d_k, v_k) \in F_{\sigma_k}(\Delta, \Delta)$ as required. □

5.4.3 Properties of the logical relations

Theorem 5.4.11 (Fundamental theorem of the logical relations). *For typing contexts Γ, values v, frame stacks S and expressions e then*

$$\begin{cases} \Gamma \vdash v : \tau \Rightarrow \forall \rho \lhd_\Gamma \psi.\ \mathcal{V}[\![\Gamma \vdash v : \tau]\!](\rho) \lhd_\tau^{\mathrm{val}} v[\psi]; \\ \Gamma \vdash_s S : \tau \multimap \tau' \Rightarrow \forall \rho \lhd_\Gamma \psi.\ \mathcal{S}[\![\Gamma \vdash_s S : \tau \multimap \tau']\!](\rho) \lhd_\tau^{\mathrm{stk}} S[\psi]; \\ \Gamma \vdash e : \tau \Rightarrow \forall \rho \lhd_\Gamma \psi.\ \mathcal{E}[\![\Gamma \vdash e : \tau]\!](\rho) \lhd_\tau^{\mathrm{exp}} e[\psi] \end{cases}$$

where $\rho \lhd_\Gamma \psi$ holds iff the domains of Γ, ρ and ψ are equal, $\vdash \psi : \Gamma$ and for each $x \in \mathrm{dom}(\rho)$, $\rho(x) \lhd_{\Gamma(x)}^{\mathrm{val}} \psi[x]$. We write $\forall \rho \lhd_\Gamma \psi$ to indicate universal quantification over all such $(\rho \in [\![\Gamma]\!], \psi \in \mathrm{Subst}_\Gamma)$-pairs for some particular Γ.

Proof. It is straightforward to see that the theorem holds when $\rho = \bot$. In the other cases, we proceed by induction over the axioms and rules defining $\vdash_s$ and $\vdash$. There are many cases to consider, the vast majority of which follow the same pattern. We now provide a good selection of them.

The first part of the proof concerns the cases for values.

▶ *Case (val-vid).* Follows by virtue of the assumption that $\rho \lhd_\Gamma \psi$.

▶ *Cases (val-unit) and (val-atom)* follow immediately from properties (5.1) and (5.2) respectively.

▶ *Case (val-con).* Given a judgement $\Gamma \vdash \mathtt{C}_k\mathtt{(}v_k\mathtt{)} : \delta$, which must have been derived by knowing $\Gamma \vdash v_k : \sigma_k$, we require $(i \circ \mathrm{in}_k)(\mathcal{V}[\![\Gamma \vdash v_k : \sigma_k]\!](\rho)) \lhd_\delta^{\mathrm{val}} (\mathtt{C}_k\mathtt{(}v_k\mathtt{)})[\psi]$. That is to say, $\exists d_k \in [\![\sigma_k]\!].\ (i \circ \mathrm{in}_k)(\mathcal{V}[\![\Gamma \vdash v_k : \sigma_k]\!](\rho)) = (i \circ \mathrm{in}_k)(d_k) \ \wedge\ d_k \lhd_{\sigma_k}^{\mathrm{val}} v_k[\psi]$ and we can see immediately that this follows from the induction hypothesis.

▶ *Case (val-pair).* Straightforward to deduce from the induction hypothesis and the property (5.5) of $\lhd_{\tau \times \tau'}^{\mathrm{val}}$.

▶ *Case (val-abst).* Given a valid typing judgement $\Gamma \vdash \mathtt{<<}a\mathtt{>>}v : \mathtt{<<name>>}\tau$, which must have been derived by knowing $\Gamma \vdash a : \mathtt{name}$ and $\Gamma \vdash v : \tau$, then the induction hypothesis tells us that $\mathcal{V}[\![\Gamma \vdash v : \tau]\!](\rho) \lhd_\tau^{\mathrm{val}} v[\psi]$. Thus by the equivariance of $\lhd_\tau^{\mathrm{val}}$ we obtain that for any $a' \notin \mathrm{supp}(\mathcal{V}[\![\Gamma \vdash v : \tau]\!](\rho), v[\psi])$,

$$(a\ a') \cdot \mathcal{V}[\![\Gamma \vdash v : \tau]\!](\rho) \lhd_\tau^{\mathrm{val}} (a\ a') \cdot (v[\psi]).$$

By property (5.4) we thus obtain $[a]\mathcal{V}[\![\Gamma \vdash v : \tau]\!](\rho) \lhd_{\mathtt{<<name>>}\tau}^{\mathrm{val}} \mathtt{<<}a\mathtt{>>}(v[\psi])$. It follows that $\mathcal{V}[\![\Gamma \vdash \mathtt{<<}a\mathtt{>>}v : \mathtt{<<name>>}\tau]\!] \lhd_{\mathtt{<<name>>}\tau}^{\mathrm{val}} (\mathtt{<<}a\mathtt{>>}v)[\psi]$ as required.

▶ *Case (val-fn).* Under the assumption that $\Gamma \vdash \mathtt{fun}\ f\mathtt{(}x\mathtt{)}\ \mathtt{=}\ e : \tau \to \tau'$, which must have been derived by knowing $\Gamma, f : \tau \to \tau', x : \tau \vdash e : \tau'$, it is necessary to show that

$$\mathcal{V}[\![\Gamma \vdash \mathtt{fun}\ f\mathtt{(}x\mathtt{)}\ \mathtt{=}\ e : \tau \to \tau']\!](\rho) \lhd_{\tau \to \tau'}^{\mathrm{val}} (\mathtt{fun}\ f\mathtt{(}x\mathtt{)}\ \mathtt{=}\ e)[\psi].$$

By property (5.6) it therefore suffices to show that $\Phi(\mathrm{fix}(g))$ holds, where

$$\Phi(F) \stackrel{\mathrm{def}}{\Leftrightarrow} \forall d \lhd_\tau^{\mathrm{val}} v.\ F(d) \lhd_{\tau'}^{\mathrm{exp}} (\mathtt{fun}\ f\mathtt{(}x\mathtt{)}\ \mathtt{=}\ e[\psi])(v)$$

and $g \stackrel{\text{def}}{=} \underline{\lambda} f' \in [\![\tau \to \tau']\!]. \underline{\lambda} x' \in [\![\tau]\!]. \mathcal{E}[\![\Gamma, f \mapsto \tau \to \tau', x \mapsto \tau \vdash e \,:\, \tau']\!](\rho[f \mapsto f', x \mapsto x'])$. Since $\Phi(F)$ holds just in case $F \lhd^{\text{val}}_{\tau \to \tau'}$ `fun f(x) = e`$[\psi]$, then $\{ F \mid \Phi(F)\}$ is an admissible subset of $[\![\tau \to \tau']\!]$ (by virtue of the admissibility property of $\lhd^{\text{val}}_{\tau \to \tau'}$). Thus by Scott induction we can conclude $\Phi(\text{fix}(g))$ by proving that for any $F \in [\![\tau \to \tau']\!]$, $\Phi(F) \Rightarrow \Phi(g(F))$. Write Γ' for $\Gamma, f \mapsto \tau \to \tau', x \mapsto \tau$ and take any $d \lhd^{\text{val}}_{\tau} v$. By assumption we have that $F \lhd^{\text{val}}_{\tau \to \tau'}$ `fun f(x) = e`$[\psi]$, so $\rho[f \mapsto F, x \mapsto d] \lhd_{\Gamma'} \psi$[`fun f(x) = e`$[\psi]/f, v/x]$. Since the induction hypothesis of the fundamental theorem tells us that for all $\sigma' \lhd^{\text{stk}}_{\tau'} S'$ and $\rho' \lhd_{\Gamma'} \psi'$, $\mathcal{E}[\![\Gamma' \vdash e \,:\, \tau']\!](\rho')(\sigma') = \top \Rightarrow \langle S', e[\psi'] \rangle\!\downarrow$, then we know that for all $\sigma' \lhd^{\text{stk}}_{\tau'} S'$, $\mathcal{E}[\![\Gamma' \vdash e \,:\, \tau']\!](\rho[f \mapsto F, x \mapsto d])(\sigma') = \top \Rightarrow \langle S', e[\psi,$ `fun f(x) = e`$[\psi]/f, v/x] \rangle \downarrow$. But combining this with the definition of the termination relation implies that $\Phi(g(F))$ holds.

The next part of the proof concerns the cases for frame stacks.

► *Case (stk-abst-left).* Given a valid typing judgement $\Gamma \vdash_s S \circ$ `<<[–]>>`e `: name ⊸ _`, which must have been derived from $\Gamma \vdash_s S$: `<<name>>`$\tau \multimap$ `_` and $\Gamma \vdash e \,:\, \tau$, then we have for $\rho \lhd_\Gamma \psi$ that

$$\mathcal{S}[\![\Gamma \vdash_s S \,:\, \texttt{<<name>>}\tau \multimap \texttt{\_}]\!](\rho) \lhd^{\text{stk}}_{\texttt{<<name>>}\tau} S[\psi]\text{; and} \tag{5.20}$$
$$\mathcal{E}[\![\Gamma \vdash e \,:\, \tau]\!](\rho) \lhd^{\text{exp}}_{\tau} e[\psi] \tag{5.21}$$

by the induction hypotheses. We can calculate that we must prove $\langle S[\psi] \circ$ `<<[–]>>`$e[\psi], a \rangle\!\downarrow$, and therefore it suffices to show

$$\langle S[\psi] \circ \texttt{<<}a\texttt{>>}[\text{–}], e[\psi] \rangle\!\downarrow \tag{5.22}$$

under the assumption that

$$\mathcal{S}[\![\Gamma \vdash_s S \circ \texttt{<<[–]>>}e \,:\, \texttt{name} \multimap \texttt{\_}]\!](\rho)(a) = \top \tag{5.23}$$

for any $a \in \mathbb{A}$. Expanding (5.20)–(5.23) and making use of the definition of the termination relation we obtain that for any $[a']d' \lhd^{\text{val}}_{\texttt{<<name>>}\tau}$ `<<`a''`>>`v' and any $\sigma' \lhd^{\text{stk}}_{\tau} S'$,

$$\mathcal{S}[\![\Gamma \vdash_s S \,:\, \texttt{<<name>>}\tau \multimap \texttt{\_}]\!](\rho)([a']d') = \top \Rightarrow \langle S[\psi] \circ \texttt{<<}a''\texttt{>>}[\text{–}], v' \rangle\!\downarrow; \tag{5.24}$$
$$\mathcal{E}[\![\Gamma \vdash e \,:\, \tau]\!](\rho)(\sigma') = \top \Rightarrow \langle S', e[\psi] \rangle\!\downarrow; \tag{5.25}$$
$$\mathcal{E}[\![\Gamma \vdash e \,:\, \tau]\!](\rho)(\lambda d \in [\![\tau]\!].\, \mathcal{S}[\![\Gamma \vdash_s S \,:\, \texttt{<<name>>}\tau \multimap \texttt{\_}]\!](\rho)([a]d)) = \top. \tag{5.26}$$

Now, observe that it is sufficient to set

$$\sigma' = \lambda d \in [\![\tau]\!].\, \mathcal{S}[\![\Gamma \vdash_s S \,:\, \texttt{<<name>>}\tau \multimap \texttt{\_}]\!](\rho)([a]d)$$

and $S' = S[\psi] \circ$ `<<`a`>>`$[\text{–}]$. For then, we can combine (5.25) and (5.26) to conclude (5.22) so long as $\sigma' \lhd^{\text{stk}}_{\tau} S'$. To see this, recall that given any $d \lhd^{\text{val}}_{\tau} v$ and any $a' \in \mathbb{A}$ then $(a\ a') \cdot d \lhd^{\text{val}}_{\tau} (a\ a') \cdot v$. Therefore $[a]d \lhd^{\text{val}}_{\texttt{<<name>>}\tau}$ `<<`a`>>`v which enables us to conclude by using (5.24).

► *Case (stk-abst-right).* Given a valid typing judgement $\Gamma \vdash_s S \circ$ `<<`a`>>`$[\text{–}] : \tau \multimap$ `_`, which must have been derived by knowing $\Gamma \vdash_s S \,:\,$ `<<name>>`τ and $\Gamma \vdash_s S \circ$ `<<`a`>>`$[\text{–}] \,:\, \tau \multimap$ `_`, consider some $\rho \lhd_\Gamma \psi$. Then we have by the induction hypothesis and the remaining assumption that for $[a']d' \lhd^{\text{val}}_{\texttt{<<name>>}\tau}$ `<<`a''`>>`v' and $d'' \lhd^{\text{val}}_{\tau} v''$,

$$\mathcal{S}[\![\Gamma \vdash_s S \,:\, \texttt{<<name>>}\tau]\!](\rho)([a']d') = \top \Rightarrow \langle S[\psi], \texttt{<<}a''\texttt{>>}v' \rangle\!\downarrow; \tag{5.27}$$
$$\mathcal{S}[\![\Gamma \vdash_s S \circ \texttt{<<}a\texttt{>>}[\text{–}] \,:\, \tau \multimap \texttt{\_}]\!](\rho)(d'') = \top. \tag{5.28}$$

We need to show that $\langle S[\psi] \circ$ `<<`a`>>`$[\text{–}], v'' \rangle\!\downarrow$ and so it suffices to prove:

$$\langle S[\psi], \texttt{<<}a\texttt{>>}v'' \rangle\!\downarrow\,. \tag{5.29}$$

Now, by definition of $\mathcal{S}[\![\text{–}]\!]$ then (5.28) is equivalent to

$$\mathcal{S}[\![\Gamma \vdash_s S \,:\, \texttt{<<name>>}\tau]\!](\rho)([a]d'') = \top. \tag{5.30}$$

Since $d'' \lhd^{\text{val}}_{\tau} v''$ then $[a]d'' \lhd^{\text{val}}_{\texttt{<<name>>}\tau}$ `<<`a`>>`v'' (by a similar argument to that in the previous case). Thus we have (5.29) by virtue of (5.27).

► *Case (stk-swap-1).* Given a judgement $\Gamma \vdash_s S \circ$ `swap [–], e' in e'' : name ⊸ _`, which must have been derived by knowing that $\Gamma \vdash_s S \,:\, \tau \multimap$ `_`, $\Gamma \vdash e' :$ `name` and $\Gamma \vdash e'' \,:\, \tau$, consider some $\rho \lhd_\Gamma \psi$. Then we have by the induction hypothesis that

$$\forall d \lhd^{\text{val}}_{\tau} v, \quad \mathcal{S}[\![\Gamma \vdash_s S \,:\, \tau \multimap \texttt{\_}]\!](\rho)(d) = \top \Rightarrow \langle S[\psi], v \rangle\!\downarrow; \tag{5.31}$$

$$\forall \sigma' \lhd^{\mathrm{stk}}_{\mathtt{name}} S', \quad \mathcal{E}[\![\Gamma \vdash e' \,:\, \mathtt{name}]\!](\rho)(\sigma') = \top \Rightarrow \langle S', e'[\psi]\rangle\downarrow; \tag{5.32}$$
$$\forall \sigma'' \lhd^{\mathrm{stk}}_{\tau} S'', \quad \mathcal{E}[\![\Gamma \vdash e'' \,:\, \tau]\!](\rho)(\sigma'') = \top \Rightarrow \langle S'', e''[\psi]\rangle\downarrow\,. \tag{5.33}$$

By assumption we also have

$$\mathcal{S}[\![\Gamma \vdash_s S \circ \mathtt{swap}\ [\_],\ e'\ \mathtt{in}\ e'' : \mathtt{name} \multimap \_]\!](\rho)(a) = \top \tag{5.34}$$

and we wish to show that $\langle (S \circ \mathtt{swap}\ [\_],\ e'\ \mathtt{in}\ e'')[\psi], a\rangle\downarrow$. It therefore suffices to prove

$$\langle S[\psi] \circ \mathtt{swap}\ a,\ [\_]\ \mathtt{in}\ e''[\psi], e'[\psi]\rangle\downarrow \tag{5.35}$$

by virtue of the definition of the termination relation. Expanding (5.34) we have that

$$\mathcal{E}[\![\Gamma \vdash e' \,:\, \mathtt{name}]\!](\rho)(\sigma') = \top \tag{5.36}$$

where $\sigma' \stackrel{\mathrm{def}}{=} \underline{\lambda} a' \in [\![\mathtt{name}]\!].\ \mathcal{E}[\![\Gamma \vdash e'' \,:\, \tau]\!](\rho)(\lambda d \in [\![\tau]\!].\ \mathcal{S}[\![\Gamma \vdash_s S \,:\, \tau \multimap \_]\!](\rho)((a\ a') \cdot d))$. We now see that proving

$$\sigma' \lhd^{\mathrm{stk}}_{\mathtt{name}} S[\psi] \circ \mathtt{swap}\ a,\ [\_]\ \mathtt{in}\ e''[\psi] \tag{5.37}$$

will allow us to conclude (5.35) by virtue of (5.32). So we take any $d \lhd^{\mathrm{val}}_{\mathtt{name}} v$ and prove that $\sigma'(d) = \top$ implies $\langle S[\psi] \circ \mathtt{swap}\ a,\ [\_]\ \mathtt{in}\ e''[\psi], v\rangle\downarrow$. If $d = \bot$ then this is trivial. Otherwise, write a' for d (which must equal v by property (5.2)) and assume that

$$\mathcal{E}[\![\Gamma \vdash e'' \,:\, \tau]\!](\rho)(\lambda d \in [\![\tau]\!].\ \mathcal{S}[\![\Gamma \vdash_s S \,:\, \tau \multimap \_]\!](\rho)((a\ a') \cdot d)) = \top.$$

We need to show that $\langle S[\psi] \circ \mathtt{swap}\ a,\ [\_]\ \mathtt{in}\ e''[\psi], a'\rangle\downarrow$ holds and thus it suffices to prove that

$$\langle S[\psi] \circ \mathtt{swap}\ a,\ a'\ \mathtt{in}\ [\_], e''[\psi]\rangle\downarrow\,. \tag{5.38}$$

Now, observe that this holds from (5.33) if

$$\lambda d \in [\![\tau]\!].\ \mathcal{S}[\![\Gamma \vdash_s S \,:\, \tau \multimap \_]\!](\rho)((a\ a') \cdot d) \lhd^{\mathrm{stk}}_{\tau} S[\psi] \circ \mathtt{swap}\ a,\ a'\ \mathtt{in}\ [\_]. \tag{5.39}$$

Thus we require that for any $d \lhd^{\mathrm{val}}_{\tau} v$ then

$$\mathcal{S}[\![\Gamma \vdash_s S \,:\, \tau \multimap \_]\!](\rho)((a\ a') \cdot d) = \top \Rightarrow \langle S[\psi] \circ \mathtt{swap}\ a,\ a'\ \mathtt{in}\ [\_], v\rangle\downarrow\,.$$

Therefore using the definition of the termination relation it suffices to show that

$$\mathcal{S}[\![\Gamma \vdash_s S \,:\, \tau \multimap \_]\!](\rho)((a\ a') \cdot d) = \top \Rightarrow \langle S[\psi], (a\ a') \cdot v\rangle\downarrow$$

and this follows from (5.31) by equivariance of $\lhd^{\mathrm{val}}_{\tau}$. Thus we have (5.39) which implies (5.38), which in turn implies (5.37). Then we have (5.35) as required.

► *Case (stk-swap-2).* Given a typing judgement $\Gamma \vdash_s \mathtt{swap}\ a,\ [\_]\ \mathtt{in}\ e'' : \mathtt{name} \multimap \_$, which must have been derived from $\Gamma \vdash_s S \,:\, \tau \multimap \_$ and $\Gamma \vdash e'' \,:\, \tau$, consider some $\rho \lhd_\Gamma \psi$. We have by the induction hypothesis that

$$\forall d \lhd^{\mathrm{val}}_{\tau} v.\ \mathcal{S}[\![\Gamma \vdash_s S \,:\, \tau \multimap \_]\!](\rho)(d) = \top \Rightarrow \langle S[\psi], v\rangle\downarrow; \tag{5.40}$$
$$\forall \sigma' \lhd^{\mathrm{stk}}_{\tau} S'.\ \mathcal{E}[\![\Gamma \vdash e'' \,:\, \tau]\!](\rho)(\sigma') = \top \Rightarrow \langle S', e''[\psi]\rangle\downarrow\,. \tag{5.41}$$

We also have by assumption that for any $a' \in \mathbb{A}_\bot$,

$$\mathcal{S}[\![\Gamma \vdash_s S \circ \mathtt{swap}\ a,\ [\_]\ \mathtt{in}\ e'' : \mathtt{name} \multimap \_]\!](\rho)(a') = \top \tag{5.42}$$

(which implies that $a' \neq \bot$) and we want $\langle S[\psi] \circ \mathtt{swap}\ a,\ [\_]\ \mathtt{in}\ e'', a'\rangle\downarrow$. It therefore suffices to show that

$$\langle S[\psi] \circ \mathtt{swap}\ a,\ a'\ \mathtt{in}\ [\_], e''[\psi]\rangle\downarrow\,. \tag{5.43}$$

Expanding (5.42) yields that

$$\mathcal{E}[\![\Gamma \vdash e'' \,:\, \tau]\!](\rho)(\lambda d \in [\![\tau]\!].\ \mathcal{S}[\![\Gamma \vdash_s S \,:\, \tau \multimap \_]\!](\rho)((a\ a') \cdot d)) = \top. \tag{5.44}$$

We can thus obtain (5.43) from (5.41) if we can show that

$$\lambda d \in [\![\tau]\!].\ \mathcal{S}[\![\Gamma \vdash_s S \,:\, \tau \multimap \_]\!](\rho)((a\ a') \cdot d) \lhd^{\mathrm{stk}}_{\tau} S[\psi] \circ \mathtt{swap}\ a,\ a'\ \mathtt{in}\ [\_]. \tag{5.45}$$

So take any $d \lhd^{\mathrm{val}}_{\tau} v$. From (5.40) and equivariance of $\lhd^{\mathrm{val}}_{\tau}$ we know that

$$\mathcal{S}[\![\Gamma \vdash_s S \,:\, \tau \multimap \_]\!](\rho)((a\ a') \cdot d) = \top \Rightarrow \langle S[\psi], (a\ a') \cdot v\rangle\downarrow$$

But we can see from the definitions of $\lhd^{\mathrm{stk}}_{\tau}$ and the termination relation that this is just (5.45). Thus we have (5.43) and can conclude.

► *Case (stk-swap-3).* Given a typing judgement $\Gamma \vdash_s S \circ \mathtt{swap}\ a,\ a'\ \mathtt{in}\ [\text{–}] : \tau \multimap \_$, which must have been derived by knowing $\Gamma \vdash a : \mathtt{name}$, $\Gamma \vdash a' : \mathtt{name}$ and $\Gamma \vdash_s S : \tau \multimap \_$, consider $\rho \lhd_\Gamma \psi$. Then the only non-trivial statement provided by the induction hypothesis is that for all $d \lhd^{\mathrm{val}}_{\tau} v$,

$$\mathcal{S}[\![\Gamma \vdash_s S \,:\, \tau \multimap \_]\!](\rho)(d) = \top \Rightarrow \langle S[\psi], v\rangle\downarrow . \tag{5.46}$$

We also have the assumption that for all such d and v,

$$\mathcal{S}[\![\Gamma \vdash_s S \circ \mathtt{swap}\ a,\ a'\ \mathtt{in}\ [\text{–}] : \tau \multimap \_]\!](\rho)(d) = \top$$

and expanding this yields that

$$\mathcal{S}[\![\Gamma \vdash_s S \,:\, \tau \multimap \_]\!](\rho)((a\ a') \cdot d) = \top. \tag{5.47}$$

We need to show that $\langle S[\psi] \circ \mathtt{swap}\ a,\ a'\ \mathtt{in}\ [\text{–}], v\rangle\downarrow$ holds. It thus suffices to prove $\langle S[\psi], (a\ a') \cdot v\rangle\downarrow$, which does indeed follow by combining (5.46) with (5.47) and using the equivariance property of $\lhd^{\mathrm{val}}_{\tau}$.

► *Case (stk-let-abst).* Given a judgement $\Gamma \vdash_s S \circ \mathtt{let}\ \texttt{<<}x\texttt{>>}x' = [\text{–}]\ \mathtt{in}\ e : \texttt{<<}\mathtt{name}\texttt{>>}\tau \multimap \_$, which must have been derived by knowing $\Gamma \vdash_s S \,:\, \tau' \multimap \_$ and $\Gamma, x \mapsto \mathtt{name}, x' \mapsto \tau \vdash e : \tau'$, consider $\rho \lhd_\Gamma \psi$. Then we can use the definitions of $\mathcal{S}[\![\text{–}]\!]$ and the termination relation together with the equivariance property of the latter to calculate that we must prove for all $[a_1]d \lhd^{\mathrm{val}}_{\texttt{<<}\mathtt{name}\texttt{>>}\tau} \texttt{<<}a_2\texttt{>>}v$ and any $a \in \mathbb{A} \setminus \mathrm{supp}(S, a_1, d, a_2, v, e, \rho)$ that

$$\begin{aligned}&\mathcal{E}[\![\Gamma, x \mapsto \mathtt{name}, x' \mapsto \tau \vdash e \,:\, \tau']\!](\rho[x \mapsto a, x' \mapsto (a\ a_1) \cdot d]) \\ &\qquad\qquad (\mathcal{S}[\![\Gamma \vdash_s S \,:\, \tau' \multimap \_]\!](\rho)) = \top \Rightarrow \\ &\langle S[\psi], e[\psi[a/x, (a\ a_2) \cdot v/x']]\rangle\downarrow .\end{aligned} \tag{5.48}$$

Since $[a_1]d \lhd^{\mathrm{val}}_{\texttt{<<}\mathtt{name}\texttt{>>}\tau} \texttt{<<}a_2\texttt{>>}v$ then by virtue of property (5.4) we know that $(a\ a_1) \cdot d \lhd^{\mathrm{val}}_{\tau} (a\ a_2) \cdot v$. Thus for any $\rho \lhd_\Gamma \psi$ then

$$\rho[x \mapsto a, x' \mapsto (a\ a_1) \cdot d] \lhd_{\Gamma, x \mapsto \mathtt{name}, x' \mapsto \tau} \psi[a/x, (a\ a_2) \cdot v/x']. \tag{5.49}$$

The induction hypothesis gives us the following.

$$\Gamma \vdash_s S \,:\, \tau' \multimap \_ \Rightarrow \forall \rho \lhd_\Gamma \psi.\ \mathcal{S}[\![\Gamma \vdash_s S \,:\, \tau' \multimap \_]\!](\rho) \lhd^{\mathrm{stk}}_{\tau} S[\psi] \tag{5.50}$$

$$\begin{aligned}&\Gamma, x \mapsto \mathtt{name}, x' \mapsto \tau \vdash e \,:\, \tau' \Rightarrow \\ &\quad \forall \rho \lhd_\Gamma \psi.\ \mathcal{E}[\![\Gamma, x \mapsto \mathtt{name}, x' \mapsto \tau \vdash e \,:\, \tau']\!](\rho) \lhd^{\mathrm{exp}}_{\tau} e[\psi]\end{aligned} \tag{5.51}$$

Proceeding in a similar manner to previous cases, we can now use (5.49) to satisfy the universal quantification in (5.50) so that we can combine (5.48) and (5.51) to get the result.

The final part of the proof concerns the cases for expressions which are not necessarily in canonical form. We give the three most interesting cases.

► *Case (exp-val).* Given some canonical form v and a valid typing judgement $\Gamma \vdash v \,:\, \tau$, we wish to know that for all $\rho \lhd_\Gamma \psi$, $\mathcal{E}[\![\Gamma \vdash v \,:\, \tau]\!](\rho) \lhd^{\mathrm{exp}}_{\tau} v[\psi]$. That is to say, for all $\sigma \lhd^{\mathrm{stk}}_{\tau} S$, $\mathcal{E}[\![\Gamma \vdash v \,:\, \tau]\!](\rho)(\sigma) = \top \Rightarrow \langle S, v[\psi]\rangle\downarrow$. Expanding the definition of $\mathcal{E}[\![\text{–}]\!]$ this is equivalent to

$$\sigma(\mathcal{V}[\![\Gamma \vdash v \,:\, \tau]\!](\rho)) = \top \Rightarrow \langle S, v[\psi]\rangle\downarrow . \tag{5.52}$$

Since $\sigma \lhd^{\mathrm{stk}}_{\tau} S$ then for all $v' \lhd^{\mathrm{val}}_{\tau} d'$, $\sigma(d') = \top \Rightarrow \langle S, v'\rangle\downarrow$ and so by the induction hypothesis we can set $d' = \mathcal{V}[\![\Gamma \vdash v \,:\, \tau]\!](\rho)$ and $v' = v[\psi]$ to yield (5.52).

► *Case (exp-fresh).* Given the typing judgement $\Gamma \vdash \mathtt{fresh} : \mathtt{name}$ then we require $\mathrm{new} \lhd^{\mathrm{exp}}_{\tau} \mathtt{fresh}$. That is to say for all $\sigma \lhd^{\mathrm{stk}}_{\mathtt{name}} S$, we need $\sigma(a) = \top \Rightarrow \langle S, a'\rangle\downarrow$ for $a \in \mathbb{A} \setminus \mathrm{supp}(\sigma)$ and $a' \in \mathbb{A} \setminus \mathrm{supp}(S)$. Let us set $a = a'$ and take $a \in \mathbb{A} \setminus \mathrm{supp}(\sigma, S)$. Then $\sigma(a) = \top \Rightarrow \langle S, a\rangle\downarrow \Leftrightarrow \langle S, \mathtt{fresh}\rangle\downarrow$ since $\sigma \lhd^{\mathrm{stk}}_{\mathtt{name}} S$.

► *Case (exp-abst).* Given a valid typing judgement $\Gamma \vdash \texttt{<<}e\texttt{>>}e' : \texttt{<<}\mathtt{name}\texttt{>>}\tau$, which must have been derived by knowing $\Gamma \vdash e : \mathtt{name}$ and $\Gamma \vdash e' : \tau$, consider $\rho \lhd_\Gamma \psi$. We calculate that we must prove

$$\mathcal{E}[\![\Gamma \vdash \texttt{<<}e\texttt{>>}e' : \texttt{<<}\mathtt{name}\texttt{>>}\tau]\!](\rho)(\sigma) = \top \Rightarrow \langle S \circ \texttt{<<}[\text{–}]\texttt{>>}e'[\psi], e[\psi]\rangle\downarrow$$

for all $\sigma \lhd^{\mathrm{stk}}_{\texttt{<<name>>}\tau} S$ and all $\rho \lhd_\Gamma \psi$. Expanding the definition of $\mathcal{E}[\![\text{–}]\!]$ this is:

$$\mathcal{E}[\![\Gamma \vdash e \,:\, \texttt{name}]\!](\rho)(\underline{\lambda} a \in [\![\texttt{name}]\!].\, \mathcal{E}[\![\Gamma \vdash e' \,:\, \tau]\!](\rho)(\lambda d \in [\![\tau]\!].\, \sigma([a]d))) = \top \Rightarrow \\ \langle S \circ \texttt{<<[–]>>} e'[\psi], e[\psi]\rangle\downarrow\,. \tag{5.53}$$

The induction hypothesis tells us that for all $\sigma_1 \lhd^{\mathrm{stk}}_{\texttt{name}} S_1$ and all $\sigma_2 \lhd^{\mathrm{stk}}_\tau S_2$,

$$\mathcal{E}[\![\Gamma \vdash e \,:\, \texttt{name}]\!](\rho)(\sigma_1) = \top \Rightarrow \langle S_1, e[\psi]\rangle\downarrow \text{ ; and} \tag{5.54}$$
$$\mathcal{E}[\![\Gamma \vdash e' \,:\, \tau]\!](\rho)(\sigma_2) = \top \Rightarrow \langle S_2, e'[\psi]\rangle\downarrow\,. \tag{5.55}$$

Using (5.53) and (5.54) we see that it suffices to show

$$\underline{\lambda} a \in [\![\texttt{name}]\!].\, \mathcal{E}[\![\Gamma \vdash e' \,:\, \tau]\!](\rho)(\lambda d \in [\![\tau]\!].\, \sigma([a]d)) \lhd^{\mathrm{stk}}_\tau S \circ \texttt{<<[–]>>} e'[\psi].$$

Take some $d \lhd^{\mathrm{val}}_{\texttt{name}} v$. If $d = \bot$ then the implication is trivial. Otherwise, write a for d (so $a = v$ also). Then using the definitions of the termination relation and $\lhd^{\mathrm{stk}}_\tau$ we deduce that we must prove

$$\mathcal{E}[\![\Gamma \vdash e' \,:\, \tau]\!](\rho)(\lambda d \in [\![\tau]\!].\, \sigma([a]d)) = \top \Rightarrow \langle S \circ \texttt{<<}a\texttt{>>}[\text{–}], e'[\psi]\rangle\downarrow$$

which can be concluded from (5.55) if $\lambda d \in [\![\tau]\!].\, \sigma([a]d) \lhd^{\mathrm{stk}}_\tau S \circ \texttt{<<}a\texttt{>>}[\text{–}]$ holds. Taking some $d \lhd^{\mathrm{val}}_\tau v$ we therefore need $\sigma([a]d) = \top \Rightarrow \langle S, \texttt{<<}a\texttt{>>}v\rangle\downarrow$. Let us take $d \neq \bot$, for if it is bottom then the implication is trivial. This holds since $\sigma \lhd^{\mathrm{stk}}_{\texttt{<<name>>}\tau} S$ and we can calculate[7] that $d \lhd^{\mathrm{val}}_\tau v \Rightarrow [a]d \lhd^{\mathrm{val}}_{\texttt{<<name>>}\tau} \texttt{<<}a\texttt{>>}v$. □

Corollary 5.4.12. *For typeable closed values v, frame stacks S and expressions e of type τ, $\tau \multimap \_$ and τ respectively then*

$$\mathcal{V}[\![v]\!] \lhd^{\mathrm{val}}_\tau v \quad \textit{and} \quad \mathcal{S}[\![S]\!] \lhd^{\mathrm{stk}}_\tau S \quad \textit{and} \quad \mathcal{E}[\![e]\!] \lhd^{\mathrm{exp}}_\tau e.$$

Proof. Immediate from Theorem 5.4.11. □

Lemma 5.4.13. *For a closed value v of type δ and $d \in [\![\delta]\!]$, $\mathrm{return}(d) \lhd^{\mathrm{exp}}_\delta v$ implies that $d = (i \circ \mathrm{in}_k)(d_k)$ and $v = \mathtt{C}_k\,(v_k)$ for some $1 \leqslant v_k \leqslant K$, closed v_k of type σ_k and $d_k \in [\![\sigma_k]\!]$.*

Proof. By virtue of i being an isomorphism, each $d \in [\![\delta]\!]$ may be expressed as $(i \circ \mathrm{in}_k)(d_k)$. The typing rules of Mini-FreshML mean that each v of type δ must be of the form $\mathtt{C}_{k'}\,(v_{k'})$. It remains to show that $k = k'$.

For each $1 \leqslant k \leqslant K$, construct the frame stack $S_k \,:\, \delta \multimap \_$ as follows:

$$S_k \stackrel{\mathrm{def}}{=} [] \circ \texttt{match}\ [\text{–}]\ \texttt{with}\ \cdots\ \texttt{|}\ \mathtt{C}_j\,(x_j)\ \texttt{->}\ t(j,k,x_j)\ \texttt{|}\ \cdots$$

where $t(\text{–},\text{–},\text{–})$ is defined like so, for some divergent term Ω:

$$t(j,k,x_j) \stackrel{\mathrm{def}}{=} \begin{cases} \mathtt{C}_j\,(x_j) & \text{if } j = k; \\ \Omega & \text{otherwise.} \end{cases}$$

Corollary 5.4.12 tells us that $\mathcal{S}[\![S_k]\!] \lhd^{\mathrm{stk}}_\delta S_k$. But since $\mathrm{return}(d) \lhd^{\mathrm{exp}}_\delta v$ then

$$\forall \sigma \lhd^{\mathrm{stk}}_\delta S.\, \sigma((i \circ \mathrm{in}_k)(d_k)) = \top \Rightarrow \langle S, \mathtt{C}_{k'}\,(v_{k'})\rangle\downarrow$$

and therefore $\mathcal{S}[\![S_k]\!]((i \circ \mathrm{in}_k)(d_k)) = \top \Rightarrow \langle S_k, \mathtt{C}_{k'}\,(v_{k'})\rangle\downarrow$. The left-hand side of this may be seen to hold always by expanding the definition of $\mathcal{S}[\![\text{–}]\!]$; therefore $\langle S_k, \mathtt{C}_{k'}\,(v_{k'})\rangle\downarrow$ holds. But this can only hold if $k = k'$. □

The following lemma is needed to derive some extensionality properties of Mini-FreshML which we shall come to in due course. It tells us about the relationship between $\lhd^{\mathrm{exp}}_\tau$ and $\lhd^{\mathrm{val}}_\tau$ when only canonical forms are involved.

Lemma 5.4.14 (Canonical forms in $\lhd^{\mathrm{exp}}_\tau$). *For a closed value v of type τ and $d \in [\![\tau]\!]$, $\mathrm{return}(d) \lhd^{\mathrm{exp}}_\tau v$ implies that $d \lhd^{\mathrm{val}}_\tau v$.*

[7] In a similar manner to the (val-abst) case.

Proof. In the case where $d = \bot$, the lemma is trivial. Otherwise, we assume $d \neq \bot$ and proceed by induction on the structure of the value v. A tricky proof, whose multiple nested implications demand great care.

► *Cases (unit)–(name)* are trivial by virtue of simple properties of $\lhd^{\mathrm{val}}_{\tau}$.

► *Case (data).* Assume $\mathrm{return}(d) \lhd^{\mathrm{exp}}_{\delta} v$. Lemma 5.4.13 then tells us that d must be of the form $(i \circ \mathrm{in}_k)(d_k)$ and v must be of the form $\mathtt{C}_k\,\mathtt{(}v_k\mathtt{)}$. Therefore

$$\forall \sigma \lhd^{\mathrm{stk}}_{\delta} S.\ \sigma((i \circ \mathrm{in}_k)(d_k)) = \top \Rightarrow \langle S, \mathtt{C}_k\,\mathtt{(}v_k\mathtt{)}\rangle\!\downarrow . \tag{5.56}$$

The induction hypothesis tells us that $\mathrm{return}(d_k) \lhd^{\mathrm{exp}}_{\sigma_k} v_k \Rightarrow d_k \lhd^{\mathrm{val}}_{\sigma_k} v_k$. Since we wish to know that $d \lhd^{\mathrm{val}}_{\delta} v$ then by expanding the definition of $\lhd^{\mathrm{exp}}_{\tau}$ we can see that it suffices to show for all $\sigma' \lhd^{\mathrm{stk}}_{\sigma_k} S'$, $\sigma'(d_k) = \top \Rightarrow \langle S', v_k\rangle\!\downarrow$. Taking any such $\sigma' \lhd^{\mathrm{stk}}_{\sigma_k} S'$ and assuming that $\sigma'(d_k) = \top$, construct the frame stack $S : \delta \multimap \_$ and element $\sigma \in [\![\delta]\!]^{\bot}$ as follows:

$$\begin{array}{lcl} S & \stackrel{\mathrm{def}}{=} & S' \circ \mathtt{match\ [\text{–}]\ with\ \cdots\ |\ C}_k\mathtt{(}x_k\mathtt{)\ ->}\ x_k\ \mathtt{|}\ \cdots \\ \sigma & \stackrel{\mathrm{def}}{=} & \lambda d \in [\![\delta]\!].\ \sigma'((i \circ \mathrm{in}_k)^{-1}(d)). \end{array}$$

Calculating that $\sigma \lhd^{\mathrm{stk}}_{\delta} S$, we can deduce from (5.56) that $\langle S, \mathtt{C}_k\,\mathtt{(}v_k\mathtt{)}\rangle\!\downarrow$. But this must have been derived by knowing that $\langle S', v_k\rangle\!\downarrow$ as required.

► *Case (abst).* Assume $\mathrm{return}(d) \lhd^{\mathrm{exp}}_{\mathtt{<<name>>}\tau} v$. Since $d \neq \bot$, it is of the form $[a]d'$ with $d' \neq \bot$ and v is of the form $\mathtt{<<}a'\mathtt{>>}v'$. Therefore,

$$\forall \sigma \lhd^{\mathrm{stk}}_{\mathtt{<<name>>}\tau} S.\ \sigma([a]d') = \top \Rightarrow \langle S, \mathtt{<<}a'\mathtt{>>}v'\rangle\!\downarrow . \tag{5.57}$$

We need to show that $[a]d' \lhd^{\mathrm{val}}_{\mathtt{<<name>>}\tau} \mathtt{<<}a'\mathtt{>>}v'$. That is to say, for any $a'' \notin \mathrm{supp}(a, a', d', v')$ then $(a\ a'') \cdot d' \lhd^{\mathrm{val}}_{\tau} (a'\ a'') \cdot v'$. This can be done by applying the induction hypothesis in the form: $\mathrm{return}((a\ a'') \cdot d') \lhd^{\mathrm{exp}}_{\tau} (a'\ a'') \cdot v' \Rightarrow (a\ a'') \cdot d' \lhd^{\mathrm{val}}_{\tau} (a'\ a'') \cdot v'$ (since swapping does not affect the size of the value v'). Then by expanding the definition of $\lhd^{\mathrm{exp}}_{\tau}$ we can see it suffices to show that the following implication holds: for any $\sigma' \lhd^{\mathrm{stk}}_{\tau} S'$, then $\sigma'((a\ a'') \cdot d') = \top \Rightarrow \langle S', (a'\ a'') \cdot v'\rangle\!\downarrow$.

Assume $\sigma'(d') = \top$ and construct the stack $S : \mathtt{<<name>>}\tau \multimap \_$ together with the element $\sigma \in [\![\mathtt{<<name>>}\tau]\!]^{\bot}$ as follows[8]:

$$\begin{array}{lcl} S & \stackrel{\mathrm{def}}{=} & S' \circ \mathtt{let\ <<}x\mathtt{>>}x' \mathtt{\ =\ [\text{–}]\ in}\ x' \\ \sigma & \stackrel{\mathrm{def}}{=} & \lambda[a]d' \in [\![\mathtt{<<name>>}\tau]\!].\ \sigma'((a\ a'') \cdot d') \quad (\ a'' \notin \mathrm{supp}(\sigma, a, a', d', v')). \end{array}$$

One can calculate that $\sigma \lhd^{\mathrm{stk}}_{\mathtt{<<name>>}\tau} S$. Since $\sigma'((a\ a'') \cdot d') = \top$, then we have $\sigma([a]d') = \sigma'((a\ a'') \cdot d') = \top$. Therefore we can apply (5.57) to deduce that $\langle S, \mathtt{<<}a'\mathtt{>>}v'\rangle\!\downarrow$; that is to say, $\langle S' \circ \mathtt{let\ <<}x\mathtt{>>}x' \mathtt{\ =\ [\text{–}]\ in}\ x', \mathtt{<<}a'\mathtt{>>}v'\rangle\!\downarrow$. This must have been deduced by knowing that $\langle S', (a'\ a'') \cdot v'\rangle\!\downarrow$ (note that we can use the same a'' here by virtue of the conditions which we have placed upon it), which is just what we require.

► *Case (pair).* Assume $\mathrm{return}(d) \lhd^{\mathrm{exp}}_{\tau \times \tau'} v$. Since $d \neq \bot$, it must be of the form $\langle d_1, d_2\rangle$ with $d_1, d_2 \neq \bot$ and v must be of the form $\mathtt{(}v_1\mathtt{,}\ v_2\mathtt{)}$; we therefore have

$$\forall \sigma \lhd^{\mathrm{stk}}_{\tau \times \tau'} S.\ \sigma(\langle d_1, d_2\rangle) = \top \Rightarrow \langle S, \mathtt{(}v_1\mathtt{,}\ v_2\mathtt{)}\rangle\!\downarrow . \tag{5.58}$$

The induction hypothesis tells us that $\mathrm{return}(d_1) \lhd^{\mathrm{exp}}_{\tau} v_1 \Rightarrow d_1 \lhd^{\mathrm{val}}_{\tau} v_1$ and $\mathrm{return}(d_2) \lhd^{\mathrm{exp}}_{\tau'} v_2 \Rightarrow d_2 \lhd^{\mathrm{val}}_{\tau'} v_2$. We need to show that $d \lhd^{\mathrm{val}}_{\tau \times \tau'} v$; that is to say, $d_1 \lhd^{\mathrm{val}}_{\tau} v_1$ and $d_2 \lhd^{\mathrm{val}}_{\tau'} v_2$. The definition of $\lhd^{\mathrm{exp}}_{\tau}$ implies that it suffices to show that for all $\sigma_1 \lhd^{\mathrm{stk}}_{\tau} S_1$ and $\sigma_2 \lhd^{\mathrm{stk}}_{\tau'} S_2$, $\sigma_1(d_1) = \top \Rightarrow \langle S_1, v_1\rangle\!\downarrow$ and $\sigma_2(d_2) = \top \Rightarrow \langle S_2, v_2\rangle\!\downarrow$.

So assume that for $\sigma_1 \lhd^{\mathrm{stk}}_{\tau} S_1$ and $\sigma_2 \lhd^{\mathrm{stk}}_{\tau'} S_2$, $\sigma_1(d_1) = \top$ and $\sigma_2(d_2) = \top$. Now construct the following stacks S, $S' : \tau \times \tau' \multimap \_$ and elements σ, $\sigma' \in [\![\tau \times \tau]\!]^{\bot}$:

$$\begin{array}{lcl} S & \stackrel{\mathrm{def}}{=} & S_1 \circ \mathtt{let\ (}x\mathtt{,}\ x'\mathtt{)\ =\ [\text{–}]\ in}\ x \\ S' & \stackrel{\mathrm{def}}{=} & S_2 \circ \mathtt{let\ (}x\mathtt{,}\ x'\mathtt{)\ =\ [\text{–}]\ in}\ x' \\ \sigma & \stackrel{\mathrm{def}}{=} & \underline{\lambda}\langle d_1, d_2\rangle \in [\![\tau \times \tau']\!].\ d_1 \end{array}$$

[8] σ is well-defined for the same reasons as $\mathtt{new}$ (p.56).

$$\sigma' \stackrel{\text{def}}{=} \underline{\lambda}\langle d_1, d_2\rangle \in [\![\tau \times \tau']\!].\ d_2.$$

One can calculate from these definitions that $\sigma \lhd^{\text{stk}}_{\tau\times\tau'} S$ and $\sigma' \lhd^{\text{stk}}_{\tau\times\tau'} S'$. Since $d_2 \neq \bot$ then $\sigma\langle d_1, d_2\rangle = \sigma_1(d_1) = \top$. It follows from (5.58) that $\langle S, (v_1\,,\ v_2)\rangle\downarrow$. But this must have been derived by knowing that $\langle S_1, v_1\rangle\downarrow$. A similar argument applies with σ' and S' to deduce that $\langle S_2, v_2\rangle\downarrow$ as required.

► *Case (fun).* Assume $\text{return}(d) \lhd^{\text{exp}}_{\tau\to\tau'} v$. Then v must be of the form `fun f(x) = e`. We therefore have

$$\forall\sigma \lhd^{\text{stk}}_{\tau\to\tau'} S.\ \sigma(d) = \top \Rightarrow \langle S, \texttt{fun}\ f\texttt{(}x\texttt{)}\ \texttt{=}\ e\rangle\downarrow\ . \tag{5.59}$$

We wish to know that $d \lhd^{\text{val}}_{\tau\to\tau'} v$; that is to say, for all $d' \lhd^{\text{val}}_{\tau} v'$ then $d(d') \lhd^{\text{exp}}_{\tau'} v\ v'$. This in turn holds just when for all $\sigma' \lhd^{\text{stk}}_{\tau'} S'$, $d(d')(\sigma') = \top \Rightarrow \langle S', v\ v'\rangle\downarrow$. Choosing any $d' \lhd^{\text{val}}_{\tau} v'$ and $\sigma' \lhd^{\text{stk}}_{\tau'} S'$ then define the stack $S : (\tau \to \tau') \multimap \_$ and the element $\sigma \in [\![\tau \to \tau']\!]^{\perp}$ as follows:

$$\begin{array}{rcl} S & \stackrel{\text{def}}{=} & S' \circ [\text{–}]\ v' \\ \sigma & \stackrel{\text{def}}{=} & \lambda d \in [\![\tau \to \tau']\!].\ d(d')(\sigma'). \end{array}$$

We calculate that $\sigma \lhd^{\text{stk}}_{\tau\to\tau'} S$. Since we assumed $d(d')(\sigma') = \top$ then $\sigma(d) = \top$; thus (5.59) gives $\langle S' \circ [\text{–}]\ v', \texttt{fun}\ f\texttt{(}x\texttt{)}\ \texttt{=}\ e\rangle\downarrow$. This implies that $\langle S', v\ v'\rangle\downarrow$ as required. □

Definition 5.4.15 (The relations $\leqslant_{\text{v}}$ and $\leqslant_{\text{e}}$). The type-respecting relation $\leqslant_{\text{e}}$, written $\Gamma \vdash e \leqslant_{\text{e}} e' : \tau$, holds iff the expressions e and e' may both be assigned type τ in typing context Γ and for all $\rho \lhd_\Gamma \psi$, $\mathcal{E}[\![\Gamma \vdash e : \tau]\!](\rho) \lhd^{\text{exp}}_{\tau} e'[\psi]$. We write $e \leqslant_{\text{e}} e'$ iff e and e' are closed expressions in the relation.

The type-respecting relation $\leqslant_{\text{v}}$, written $\Gamma \vdash v \leqslant_{\text{v}} v' : \tau$, holds iff the expressions in canonical form v and v' may both be assigned type τ in typing context Γ and for all $\rho \lhd_\Gamma \psi$ then $\mathcal{V}[\![\Gamma \vdash v : \tau]\!](\rho) \lhd^{\text{val}}_{\tau} v'[\psi]$. We write $v \leqslant_{\text{v}} v'$ iff v and v' are closed expressions in the relation. ◇

Lemma 5.4.16 ($\leqslant_{\text{v}}$ contained within $\leqslant_{\text{e}}$). *For expressions in canonical form v and v' which may be assigned type τ in typing context Γ then*

$$\Gamma \vdash v \leqslant_{\text{v}} v' : \tau \Rightarrow \Gamma \vdash v \leqslant_{\text{e}} v' : \tau.$$

Proof. It suffices to show that for all $\rho \lhd_\Gamma \psi$,

$$\mathcal{V}[\![\Gamma \vdash v : \tau]\!](\rho) \lhd^{\text{val}}_{\tau} v'[\psi] \Rightarrow \lambda\sigma \in [\![\tau]\!]^{\perp}.\ \sigma(\mathcal{V}[\![\Gamma \vdash v : \tau]\!](\rho)) \lhd^{\text{exp}}_{\tau} v'[\psi].$$

By definition of $\lhd^{\text{exp}}_{\tau}$ this is equivalent to asking that

$$\mathcal{V}[\![\Gamma \vdash v : \tau]\!](\rho) \lhd^{\text{val}}_{\tau} v'[\psi] \Rightarrow \forall\sigma \lhd^{\text{stk}}_{\tau} S.\ \sigma(\mathcal{V}[\![\Gamma \vdash v : \tau]\!](\rho)) = \top \Rightarrow \langle S, v'[\psi]\rangle\downarrow$$

which holds by virtue of the definition of $\lhd^{\text{stk}}_{\tau}$. □

Lemma 5.4.17 (Compositionality). *For typeable expressions e, e' and contexts C such that $C[e]$ and $C[e']$ are also typeable then*

$$\Gamma \vdash e \leqslant_{\text{e}} e' : \tau \Rightarrow \Gamma \vdash C[e] \leqslant_{\text{e}} C[e'] : \tau'.$$

Proof. The lemma is proved by induction on the structure of the context C. We give just a single case here to illustrate the logic behind the proof.

► *Case (abst).* For contexts C and C' we wish to show that $\Gamma \vdash e \leqslant_{\text{e}} e' : \tau \Rightarrow \Gamma \vdash$ $(\texttt{<<}C\texttt{>>}C')[e] \leqslant_{\text{e}} (\texttt{<<}C\texttt{>>}C')[e']$: $\texttt{<<name>>}\tau'$. That is to say, $\Gamma \vdash e \leqslant_{\text{e}} e' : \tau \Rightarrow \Gamma \vdash$ $\texttt{<<}C[e]\texttt{>>}C'[e] \leqslant_{\text{e}} \texttt{<<}C[e']\texttt{>>}C'[e']$: $\texttt{<<name>>}\tau'$. As induction hypotheses we have

$$\Gamma \vdash e \leqslant_{\text{e}} e' : \tau \Rightarrow \Gamma \vdash C[e] \leqslant_{\text{e}} C[e'] : \texttt{name}; \tag{5.60}$$
$$\Gamma \vdash e \leqslant_{\text{e}} e' : \tau \Rightarrow \Gamma \vdash C'[e] \leqslant_{\text{e}} C'[e'] : \tau' \tag{5.61}$$

and as further assumptions we have that for all $\rho \lhd_\Gamma \psi$ and $\sigma \lhd^{\text{stk}}_{\texttt{<<name>>}\tau'} S$,

$$\Gamma \vdash e \leqslant_{\text{e}} e' : \tau; \tag{5.62}$$
$$\mathcal{E}[\![\Gamma \vdash \texttt{<<}C[e]\texttt{>>}C'[e] : \texttt{<<name>>}\tau']\!](\rho)(\sigma) = \top. \tag{5.63}$$

Expanding (5.63) then we obtain

$$\mathcal{E}[\![\Gamma \vdash C[e] \ : \ \mathtt{name}]\!](\rho)(\lambda a \in [\![\mathtt{name}]\!].\ \mathcal{E}[\![\Gamma \vdash C'[e] \ : \ \tau']\!](\rho)(\lambda d \in [\![\tau']\!].\ \sigma([a]d))) = \top.$$

We need to show that $\langle S, (\texttt{<<}C[e']\texttt{>>}C'[e'])[\psi]\rangle\downarrow$. It therefore suffices to show that $\langle S \circ \texttt{<<}[\text{–}]\texttt{>>}(C'[e'])[\psi], (C[e'])[\psi]\rangle\downarrow$. This follows from (5.60), (5.62) and 5.64 so long as $\underline{\lambda} a \in [\![\mathtt{name}]\!].\ \mathcal{E}[\![\Gamma \vdash C'[e] \ : \ \tau']\!](\rho)(\lambda d \in [\![\tau']\!].\ \sigma([a]d)) \lhd^{\mathrm{stk}}_{\mathtt{name}} S \circ \texttt{<<}[\text{–}]\texttt{>>}C'[e'][\psi]$. To see this, we use the definition of $\lhd^{\mathrm{stk}}_{\mathtt{name}}$: take any $a \in \mathbb{A}$ (for the bottom case is trivial) and assume

$$\mathcal{E}[\![\Gamma \vdash C'[e] \ : \ \tau']\!](\rho)(\lambda d \in [\![\tau']\!].\ \sigma([a]d)) = \top. \tag{5.64}$$

We need to prove that $\langle S \circ \texttt{<<}[\text{–}]\texttt{>>}C'[e'][\psi], a\rangle\downarrow$. It therefore suffices to show that $\langle S \circ \texttt{<<}a\texttt{>>}[\text{–}], C'[e'][\psi]\rangle\downarrow$ holds. We can prove this from (5.61), (5.62) and (5.64) by showing that $\lambda d \in [\![\tau']\!].\ \sigma([a]d) \lhd^{\mathrm{stk}}_{\tau} S \circ \texttt{<<}[\text{–}]\texttt{>>}C'[e'][\psi]$. Taking any $d \lhd^{\mathrm{val}}_{\tau'} v$ and expanding the definition of $\lhd^{\mathrm{stk}}_{\tau}$, we have to show that $\langle S \circ \texttt{<<}a\texttt{>>}[\text{–}], v\rangle\downarrow$ holds given that $\sigma([a]d) = \top$. It therefore suffices to show that $\langle S, \texttt{<<}a\texttt{>>}v\rangle\downarrow$, which holds since $\sigma \lhd^{\mathrm{stk}}_{\tau} S$ and $[a]d \lhd^{\mathrm{val}}_{\texttt{<<name>>}\tau'} \texttt{<<}a\texttt{>>}v$. □

5.4.4 Completing the proof

We are now in a position to state the proof of the 'computational adequacy' result, which we recall was as follows:
Given a typing context Γ *and a typeable expression* e *such that* $\Gamma \vdash e \ : \ \tau$*, then for all closed frame stacks* S *of type* $\tau \multimap \ \_$ *and substitutions*[9] $\psi \in \mathrm{Subst}_\Gamma$

$$\mathcal{E}[\![\Gamma \vdash e \ : \ \tau]\!]\mathcal{V}[\![\psi]\!]\mathcal{S}[\![S]\!] = \top \Leftrightarrow \langle S, e[\psi]\rangle\downarrow . \tag{5.65}$$

Thus if e *is a closed expression we have that*

$$\mathcal{E}[\![e]\!]\mathcal{S}[\![S]\!] = \top \Leftrightarrow \langle S, e\rangle\downarrow . \quad \diamond \tag{5.66}$$

Proof. We start by proving (5.66), for having established this then we can deduce that for expressions e and substitutions ψ as above, $\mathcal{E}[\![e[\psi]]\!]\mathcal{S}[\![S]\!] = \top \Leftrightarrow \langle S, e[\psi]\rangle\downarrow$. Then we can apply Lemma 5.3.4 to obtain (5.65).

Now the forwards direction of (5.66) follows immediately by combining Corollary 5.4.12 first with (5.7) and secondly with (5.8). For the reverse direction (the 'soundness' property of the denotational semantics) we show that the property

$$\Phi(S, e) \stackrel{\mathrm{def}}{=} \langle S, e\rangle\downarrow \Rightarrow \mathcal{E}[\![e]\!]\mathcal{S}[\![S]\!] = \top$$

is closed under the axiom and rules defining the termination relation (Figure 3.7). The induction splits into two halves, paralleling the construction of the termination relation. For the first part, we take e to be a canonical form (which we call v for clarity) and consider the possible cases for S: here we give the interesting cases.

► *Case (empty).* In this case, $\langle [], v\rangle\downarrow$ always holds; moreover, $\mathcal{E}[\![v]\!]\mathcal{S}[\![[]]\!]$ is equal to $\mathcal{S}[\![[]]\!]\mathcal{V}[\![v]\!]$ which in turn is always $\top$.

► *Case (con).* We wish to show $\Phi(S \circ \mathtt{C}_k([\text{–}]), v)$. By the induction hypothesis we know $\langle S, \mathtt{C}_k(v)\rangle\downarrow \Rightarrow \mathcal{E}[\![\mathtt{C}_k(v)]\!]\mathcal{S}[\![S]\!] = \top$ and by assumption we have that $\langle S \circ \mathtt{C}_k([\text{–}]), v\rangle\downarrow$. Considering the derivation of this, we must have that $\mathcal{E}[\![\mathtt{C}_k(v)]\!]\mathcal{S}[\![S]\!] = \top$ and therefore (by definition) $\mathcal{S}[\![S]\!]((i \circ \mathrm{in}_k)(\mathcal{V}[\![v]\!])) = \top$. However this is just $\mathcal{S}[\![S \circ \mathtt{C}_k([\text{–}])]\!](v)$ by definition of $\mathcal{S}[\![\text{–}]\!]$; therefore, $\mathcal{E}[\![v]\!]\mathcal{S}[\![S \circ \mathtt{C}_k([\text{–}])]\!] = \top$.

► *Case (pair-l).* We wish to show $\Phi(S \circ (\texttt{[–]}, \ e), v)$. The induction hypothesis tells us that $\langle S \circ (v, \ \texttt{[–]}), e\rangle\downarrow \Rightarrow \mathcal{E}[\![e]\!]\mathcal{S}[\![S \circ (v, \ \texttt{[–]})]\!] = \top$ and we have by assumption that $\langle S \circ (\texttt{[–]}, \ e), v\rangle\downarrow$. This must have been derived by knowing that $\langle S \circ (v, \ \texttt{[–]}), e\rangle\downarrow$. We need that $\mathcal{E}[\![v]\!]\mathcal{S}[\![S \circ (\texttt{[–]}, \ e)]\!] = \top$, i.e. $\mathcal{E}[\![e]\!](\lambda d' \in [\![\tau']\!].\ \mathcal{S}[\![S]\!]\langle\mathcal{V}[\![v]\!], d'\rangle) = \top$. But this is just $\mathcal{E}[\![e]\!]\mathcal{S}[\![S \circ (v, \ \texttt{[–]})]\!]$ and so we are done.

► *Case (pair-r).* We wish to show $\Phi(S \circ (v, \ \texttt{[–]}), v')$. By the induction hypothesis we have that $\langle S, (v, \ v')\rangle\downarrow \Rightarrow \mathcal{E}[\![(v, \ v')]\!]\mathcal{S}[\![S]\!] = \top$. The right-hand side of this holds by virtue of the assumption (in a similar manner to the previous case); expanding this yields that

[9] Recall that ψ maps value identifiers to closed values; thus, $e[\psi]$ must be a closed expression since e is typeable in context Γ.

$\mathcal{S}[\![S]\!]\langle\mathcal{V}[\![v]\!], \mathcal{V}[\![v']\!]\rangle = \top$. We wish to show that $\mathcal{V}[\![v']\!]\mathcal{S}[\![S \circ (v,\ [\text{–}])]\!] = \top$, i.e. $\mathcal{S}[\![S]\!]\langle\mathcal{V}[\![v]\!], \mathcal{V}[\![v']\!]\rangle = \top$. But we already have this from above.

▶ *Cases (abst-l) and (abst-r)* argue in the same way as for (pair-l) and (pair-r), *mutatis mutandis*.

▶ *Case (swap-1).* We wish to show that $\Phi(S \circ \texttt{swap}\ [\text{–}],\ e'\ \texttt{in}\ e'', v)$ holds. Proceeding in a similar manner to the previous case, we obtain from the induction hypothesis that $\mathcal{E}[\![e']\!]\mathcal{S}[\![S \circ \texttt{swap}\ v,\ [\text{–}]\ \texttt{in}\ e'']\!] = \top$. We need to show that $\mathcal{E}[\![v]\!]\mathcal{S}[\![S \circ \texttt{swap}\ [\text{–}],\ e'\ \texttt{in}\ e'']\!] = \top$, the left-hand side of which is just $\mathcal{E}[\![e']\!]\mathcal{S}[\![S \circ \texttt{swap}\ v,\ [\text{–}]\ \texttt{in}\ e'']\!]$. This is equal to $\top$ from above.

▶ *Case (swap-2).* Proceeds in a similar manner to the previous case; we omit the details.

▶ *Case (app-l).* We wish to show that $\Phi(S \circ [\text{–}]\ e', v)$ holds; we thus require that under the assumption $\langle S \circ [\text{–}]\ e', v\rangle\!\downarrow$ then $\mathcal{E}[\![v]\!]\mathcal{S}[\![S \circ [\text{–}]\ e']\!] = \top$. This is just $\mathcal{E}[\![e']\!]\mathcal{S}[\![S \circ v\ [\text{–}]]\!] = \top$ which again holds by virtue of the induction hypothesis.

▶ *Case (app-r).* A more interesting case to break the monotony. We wish to show that $\Phi(S \circ v\ [\text{–}], v')$ holds. We assume that $\langle S \circ v\ [\text{–}], v'\rangle\!\downarrow$, where v is of the form $\texttt{fun}\ f\texttt{(}x\texttt{)}\ \texttt{=}\ e$ and of type $\tau \to \tau'$. By considering how this must have been derived, the induction hypothesis yields that $\mathcal{E}[\![e[v/f, v'/x]]\!]\mathcal{S}[\![S]\!] = \top$. We need to prove that $\mathcal{E}[\![v']\!]\mathcal{S}[\![S \circ v\ [\text{–}]]\!] = \top$, i.e. $(\mathcal{V}[\![v]\!]\ \mathcal{V}[\![v']\!])\mathcal{S}[\![S]\!] = \top$. By expanding the definition of $\mathcal{V}[\![\text{–}]\!]$ this is

$$\mathcal{E}[\![\{f\ :\ \tau \to \tau', x\ :\ \tau\} \vdash e\ :\ \tau']\!]\{f \mapsto \mathcal{V}[\![v]\!], x \mapsto \mathcal{V}[\![v']\!]\}\mathcal{S}[\![S]\!] = \top.$$

Now we can apply the substitutivity property of the denotational semantics (Lemma 5.3.4) to deduce that this equivalent to $\mathcal{E}[\![e[v/f, v'/x]]\!]\mathcal{S}[\![S]\!] = \top$, which holds from the induction hypothesis.

▶ *Case (let).* We wish to prove that $\Phi(S \circ \texttt{let}\ x\ \texttt{=}\ [\text{–}]\ \texttt{in}\ e', v)$, where $\vdash v\ :\ \tau'$ and $\{x\ :\ \tau'\} \vdash e'\ :\ \tau$. We assume that $\langle S \circ \texttt{let}\ x\ \texttt{=}\ [\text{–}]\ \texttt{in}\ e', v\rangle\!\downarrow$ holds; this must have been derived by knowing that $\langle S, e'[v/x]\rangle\!\downarrow$. We can then apply the induction hypothesis to deduce that $\mathcal{E}[\![e']\!]\mathcal{S}[\![S]\!] = \top$. We need to prove that $\mathcal{E}[\![v]\!]\mathcal{S}[\![S \circ \texttt{let}\ x\ \texttt{=}\ [\text{–}]\ \texttt{in}\ e']\!] = \top$. Expanding the left-hand side of this yields $\mathcal{E}[\![\{x\ :\ \tau'\} \vdash e'\ :\ \tau]\!]\mathcal{V}[\![v]\!]$. Applying Lemma 5.3.4 once again, we see that this is equivalent to $\mathcal{E}[\![e'[v/x]]\!]$. This is equal to $\top$ from the induction hypothesis.

▶ *Case (let-pair).* As for (let-abst), *mutatis mutandis*.

▶ *Case (let-abst).* Given $\vdash v\ :\ \texttt{<<name>>}\tau'$ and $\{x\ :\ \texttt{name}, x'\ :\ \tau'\} \vdash e'\ :\ \tau$ then we wish to prove that $\Phi(S \circ \texttt{let <<}x\texttt{>>}x'\ \texttt{=}\ [\text{–}]\ \texttt{in}\ e', v)$. We therefore have the assumption $\langle S \circ \texttt{let <<}x\texttt{>>}x'\ \texttt{=}\ [\text{–}]\ \texttt{in}\ e', v\rangle\!\downarrow$ which must have been derived by knowing that $\langle S, e'[a/x, ((a\ a') \cdot v')/x']\rangle\!\downarrow$, where $v = \texttt{<<}a\texttt{>>}v'$ and $a' \in \mathbb{A} \setminus \mathrm{supp}(S, a, v, e')$. We can thus apply the induction hypothesis to deduce that

$$\mathcal{E}[\![e'[a/x, ((a\ a') \cdot v')/x']]\!]\mathcal{S}[\![S]\!] = \top. \tag{5.67}$$

What we want to know is $\mathcal{E}[\![\texttt{<<}a\texttt{>>}v']\!]\mathcal{S}[\![S \circ \texttt{let <<}x\texttt{>>}x'\ \texttt{=}\ [\text{–}]\ \texttt{in}\ e']\!] = \top$. Expanding the left-hand side of this yields

$$\mathcal{E}[\![\{x\ :\ \texttt{name}, x'\ :\ \tau'\} \vdash e'\ :\ \tau]\!]\{x \mapsto a'', x' \mapsto (a\ a'') \cdot \mathcal{V}[\![v']\!]\}\mathcal{S}[\![S]\!]$$

for any atom $a'' \in \mathbb{A} \setminus \mathrm{supp}(S, a, v', e')$. Applying Lemma 5.3.4 once again, we see that this is equivalent to $\mathcal{E}[\![e'[a''/x, (a\ a'') \cdot v']]\!]\mathcal{S}[\![S]\!]$. Now, we can always choose a' to be the same as a'' (by selecting one which lies in $\mathbb{A} \setminus \mathrm{supp}(S, a, v, v', e')$)—meaning that we can conclude by virtue of (5.67).

▶ *Case (match).* We wish to show

$$\Phi(S \circ \texttt{match}\ [\text{–}]\ \texttt{with}\ \cdots\ \texttt{|}\ \texttt{C}_k\texttt{(}x_k\texttt{)}\ \texttt{->}\ e_k\ \texttt{|}\ \cdots, v)$$

holds. By the induction hypothesis we know that there exists some $1 \leqslant k \leqslant K$ such that

$$\langle S, e_k[v_k/x_k]\rangle\!\downarrow \Rightarrow \mathcal{E}[\![e_k[v_k/x_k]]\!]\mathcal{S}[\![S]\!] = \top, \tag{5.68}$$

where v is of the form $\texttt{C}_k\texttt{(}v_k\texttt{)}$. Now, given the assumption

$$\langle S \circ \texttt{match}\ [\text{–}]\ \texttt{with}\ \cdots\ \texttt{|}\ \texttt{C}_k\texttt{(}x_k\texttt{)}\ \texttt{->}\ e_k\ \texttt{|}\ \cdots, v\rangle\!\downarrow$$

then the left-hand side of (5.68) must hold in the derivation from the termination relation. It therefore suffices to prove that $\mathcal{E}[\![e_k[v_k/x_k]]\!]\mathcal{S}[\![S]\!] = \top$ implies

$$\mathcal{E}[\![v]\!]\mathcal{S}[\![S \circ \texttt{match}\ [\text{–}]\ \texttt{with}\ \cdots\ \texttt{|}\ \texttt{C}_k\texttt{(}x_k\texttt{)}\ \texttt{->}\ e_k\ \texttt{|}\ \cdots]\!] = \top.$$

Expanding the definitions of $\mathcal{E}[\![\text{–}]\!]$ and $\mathcal{S}[\![\text{–}]\!]$, this is equivalent to

$$\mathcal{E}[\![\{x_k : \sigma_k\} \vdash e_k : \tau]\!]\{x_k \mapsto d_k\}\mathcal{S}[\![S]\!] = \top \tag{5.69}$$

for the unique d_k such that $(i \circ \mathrm{in}_k)(d_k) = \mathcal{V}[\![v]\!]$. But since v is of the form $\mathtt{C}_k(v_k)$, then $(i \circ \mathrm{in}_k)(d_k) = \mathcal{V}[\![\mathtt{C}_k(v_k)]\!] = (i \circ \mathrm{in}_k)\mathcal{V}[\![v_k]\!]$. Thus $d_k = \mathcal{V}[\![v_k]\!]$ and so we can apply Lemma 5.3.4 to deduce that (5.69) is equivalent to $\mathcal{E}[\![e_k[v_k/x_k]]\!]\mathcal{S}[\![S]\!] = \top$, which holds by assumption.

For the second part of the proof, we take e to be non-canonical. These cases are more straightforward[10] and we only provide some salient ones.

▶ *Case (nc-con).* We wish to show $\Phi(S, \mathtt{C}_k(e))$. Calculate from the induction hypothesis and our assumption $\langle S, \mathtt{C}_k(e)\rangle\downarrow$ that $\mathcal{E}[\![e]\!]\mathcal{S}[\![S \circ \mathtt{C}_k([\text{–}])]\!] = \top$. It is necessary to show that $\mathcal{E}[\![\mathtt{C}_k(e)]\!]\mathcal{S}[\![S]\!] = \top$; this follows immediately from the previous sentence and the definition of $\mathcal{E}[\![\text{–}]\!]$.

▶ *Case (nc-fresh).* We wish to show that $\Phi(S, \mathtt{fresh})$ holds. Consequently we have the assumption that $\langle S, \mathtt{fresh}\rangle\downarrow$; this must have been derived by knowing that $\langle S, a\rangle\downarrow$ for some a not in the support of S. We need that $\mathcal{E}[\![\mathtt{fresh}]\!]\mathcal{S}[\![S]\!] = \top$, i.e. $\mathcal{S}[\![S]\!](a) = \top$ for similar a. But the induction hypothesis tells us that $\mathcal{E}[\![a]\!]\mathcal{S}[\![S]\!] = \top$, which is just $\mathcal{S}[\![S]\!](a) = \top$ as required.

▶ *Cases (nc-pair) and (nc-app)* are as for case (abst), *mutatis mutandis*.

▶ *Case (nc-abst).* We wish to show that $\Phi(S, \texttt{<<}e\texttt{>>}e')$ holds. Assuming that $\langle S, \texttt{<<}e\texttt{>>}e'\rangle\downarrow$ then the induction hypothesis yields $\mathcal{E}[\![e]\!]\mathcal{S}[\![S \circ \texttt{<<}[\text{–}]\texttt{>>}e']\!] = \top$. We wish to know $\mathcal{E}[\![\texttt{<<}e\texttt{>>}e']\!]\mathcal{S}[\![S]\!] = \top$, which immediately expands to this equality.

▶ *Case (nc-swap).* We need that $\Phi(S, \mathtt{swap}\ e,\ e'\ \mathtt{in}\ e'')$. Assume $\langle S, \mathtt{swap}\ e,\ e'\ \mathtt{in}\ e''\rangle\downarrow$. The induction hypothesis gives that

$$\mathcal{E}[\![e]\!]\mathcal{S}[\![\mathtt{swap}\ [\text{–}],\ e'\ \mathtt{in}\ e'']\!] = \top.$$

We need $\mathcal{E}[\![\mathtt{swap}\ e,\ e'\ \mathtt{in}\ e'']\!]\mathcal{S}[\![S]\!] = \top$; however, this is equivalent to the previous statement just by expanding the definitions of $\mathcal{E}[\![\text{–}]\!]$ and $\mathcal{S}[\![\text{–}]\!]$. □

5.5 The road to equivalence

In this section we are going to derive some properties of the denotational semantics which will enable us to connect with the notions of observational equivalence from Chapter 3. The first task is to show that CIU-equivalence and contextual equivalence coincide for Mini-FreshML. In order to do this, we go via the families of logical relations.

Lemma 5.5.1 (CIU-preorder coincides with $\leqslant_{\mathrm{e}}$). *For expressions e, e' then $\Gamma \vdash e \prec_{\mathrm{ciu}} e' : \tau \Leftrightarrow \Gamma \vdash e \leqslant_{\mathrm{e}} e' : \tau$.*

Proof. For the forwards direction, by definition of $\prec_{\mathrm{ciu}}$ and $\lhd_\tau^{\mathrm{exp}}$ we have as assumptions that

$$\forall S : \tau \multimap \_, \psi \in \mathrm{Subst}_\Gamma.\ \langle S, e[\psi]\rangle\downarrow \Rightarrow \langle S, e'[\psi]\rangle\downarrow\ ; \tag{5.70}$$
$$\text{for all } \sigma \lhd_\tau^{\mathrm{stk}} S', \mathcal{E}[\![e]\!](\sigma) = \top. \tag{5.71}$$

By Corollary 5.4.12 we have that $\mathcal{S}[\![S]\!] \lhd_\tau^{\mathrm{stk}} S$ for each S of type $\tau \multimap \_$. The result then follows immediately from (5.70), (5.71) and the computational adequacy result (Theorem 5.4.1). The reverse direction is similarly straightforward. □

Lemma 5.5.2 ($\approx_{\mathrm{ciu}}$ contained within $\approx_{\mathrm{ctx}}$). *The relation $\prec_{\mathrm{ciu}}$ is contained within $\prec_{\mathrm{ctx}}$, so each CIU-equivalence is also a contextual equivalence.*

Proof. It obviously suffices to consider the pre-orders. Take expressions e, e' such that $\Gamma \vdash e \prec_{\mathrm{ciu}} e' : \tau$. By Lemma 5.5.1 we then deduce $\Gamma \vdash e \leqslant_{\mathrm{e}} e' : \tau$. Given any closing context C such that $C[e]$ and $C[e']$ are well-typed then we wish to show that $\langle [], C[e]\rangle\downarrow$ implies $\langle [], C[e']\rangle\downarrow$. Assuming $\langle [], C[e]\rangle\downarrow$ holds, we know by adequacy (Theorem 5.4.1) that $\mathcal{E}[\![C[e]]\!]\mathcal{S}[\![[]]\!] = \top$. Since $\leqslant_{\mathrm{e}}$ is a congruence (Lemma 5.4.17) we also have that $C[e] \leqslant_{\mathrm{e}} C[e']$. Combining these two facts with property (5.8) yields that $\langle [], C[e']\rangle\downarrow$ as required. □

[10] Basically because if viewing the termination relation from an 'abstract machine' perspective, the actions taken upon encountering a non-canonical form do not vary a great deal. The vast majority of them consist of simply picking the first sub-expression to be evaluated and considering the termination of that in a frame stack directly derived from the original expression.

We aim to show that $\prec_{\mathrm{ctx}}$ is contained within $\prec_{\mathrm{ciu}}$ to enable us to deduce that the two relations coincide. To do this we must make use of Lemma 5.5.2 to prove the following substitutivity property of contextual equivalence.

Lemma 5.5.3. *For a typing context Γ, an identifier $x \notin \mathrm{dom}(\Gamma)$, a value v (not containing x) such that for some τ, $\vdash v : \tau$ and expressions e, e' then*

$$\Gamma, x : \tau \vdash e \prec_{\mathrm{ctx}} e' : \tau' \Rightarrow \Gamma \vdash e[v/x] \prec_{\mathrm{ctx}} e'[v/x] : \tau'.$$

Thus for substitutions ϕ with domain $X = \{x_1, \ldots, x_n\}$, a typing context Γ (with X chosen suitably so that $\mathrm{dom}(\Gamma) \cap X = \emptyset$) and expressions e, e' then

$$\Gamma, x_1 : \tau_1, \cdots, x_n : \tau_n \vdash e \prec_{\mathrm{ctx}} e' : \tau' \Rightarrow \Gamma \vdash e[\psi] \prec_{\mathrm{ctx}} e'[\psi] : \tau'.$$

where for all $1 \leqslant k \leqslant n$ we have that $\vdash \psi(x_k) : \tau_k$.

Proof. It is straightforward to see that the second sentence follows from the first. For the latter, first note that for a fresh identifier f not occurring in e or v, then $\Gamma \vdash (\mathtt{fun}\ f(x) = e)(v) \approx_{\mathrm{ciu}} e[v/x] : \tau'$ holds by virtue of the definition of $\langle -, - \rangle\!\downarrow$. Since we have established that every CIU-equivalence is also a contextual equivalence, then we also have

$$\Gamma \vdash (\mathtt{fun}\ f(x) = e)(v) \approx_{\mathrm{ctx}} e[v/x] : \tau'. \tag{5.72}$$

Now given $\Gamma, x : \tau \vdash e \prec_{\mathrm{ctx}} e' : \tau'$ then for any context C binding x, $\Gamma \vdash C[e] \prec_{\mathrm{ctx}} C[e'] : \tau$ since $\prec_{\mathrm{ctx}}$ is a congruence. Setting C to be the context $(\mathtt{fun}\ f(x) = [-])(v)$, $\Gamma \vdash (\mathtt{fun}\ f(x) = e)(v) \prec_{\mathrm{ctx}} (\mathtt{fun}\ f(x) = e')(v) : \tau$. But from (5.72) this implies that $\Gamma \vdash e[v/x] \prec_{\mathrm{ctx}} e'[v/x] : \tau$. □

Lemma 5.5.4 ($\approx_{\mathrm{ctx}}$ contained within $\approx_{\mathrm{ciu}}$). *The relation $\prec_{\mathrm{ctx}}$ is contained within $\prec_{\mathrm{ciu}}$ and thus each contextual equivalence is also a CIU-equivalence.*

Proof. Recalling the definition of $\mathcal{C}txt(-)$ from §3.4, consider two expressions e, e' related by the contextual pre-order, $\Gamma \vdash e \prec_{\mathrm{ctx}} e' : \tau$. Therefore for any context C, $\langle [], C[e] \rangle\!\downarrow$ implies that $\langle [], C[e'] \rangle\!\downarrow$. We need to show that $\Gamma \vdash e \prec_{\mathrm{ciu}} e' : \tau$, so we assume that for any frame stack S of type $\tau \multimap \_$ and closing substitution ψ then $\langle S, e[\psi] \rangle\!\downarrow$. But if this holds then the forwards direction of Lemma 3.6.3 tells us that $\langle [], (\mathcal{C}txt(S))[e[\psi]] \rangle\!\downarrow$ does also. This implies that $\langle [], (\mathcal{C}txt(S))[e'[\psi]] \rangle\!\downarrow$ (since e and e' are related by $\prec_{\mathrm{ctx}}$). Therefore $\langle S, e'[\psi] \rangle\!\downarrow$ does indeed hold by the reverse direction of Lemma 3.6.3. □

Lemmata 5.5.2 and 5.5.4 immediately give us the following.

Theorem 5.5.5 (Coincidence). *The relations of $\approx_{\mathrm{ciu}}$ and $\approx_{\mathrm{ctx}}$ coincide.* □

We are shortly going to derive some *extensionality* properties of Mini-FreshML contextual equivalence. These will enable us to take two values which are contextually equivalent and deduce properties about the contextual equivalence of their innards; or indeed the converse. In order to prove these properties, we shall require the following lemma which relates contextual preorder to the logical relations.

Lemma 5.5.6. *Given expressions e, e' and canonical forms v, v' then $\Gamma \vdash e \prec_{\mathrm{ctx}} e' : \tau \Leftrightarrow \Gamma \vdash e \leqslant_{\mathrm{e}} e' : \tau$; also, $\Gamma \vdash v \prec_{\mathrm{ctx}} v' : \tau \Leftrightarrow \Gamma \vdash v \leqslant_{\mathrm{v}} v' : \tau$.*

Proof. The first bi-implication holds by Theorem 5.5.5 and Lemma 5.5.1. For the forwards direction of the second we have that $\Gamma \vdash v \prec_{\mathrm{ctx}} v' : \tau$, which by the same Theorem and Lemma is equivalent to stating $\Gamma \vdash v \prec_{\mathrm{ciu}} v' : \tau$. This in turn implies $\Gamma \vdash v \leqslant_{\mathrm{e}} v' : \tau$ by Lemma 5.5.1. Lemma 5.4.14 then enables us to deduce that in fact $\Gamma \vdash v \leqslant_{\mathrm{v}} v' : \tau$. In the reverse direction, Lemma 5.4.16 tells us that $\Gamma \vdash v \leqslant_{\mathrm{v}} v' : \tau$ implies $\Gamma \vdash v \leqslant_{\mathrm{e}} v' : \tau$, which in turn implies $\Gamma \vdash v \prec_{\mathrm{ctx}} v' : \tau$ as previously. □

Corollary 5.5.7 (Extensionality). *For closed values v, v', etc. we have the following extensionality results.*

- ▷ *For unit values:* $\vdash v \approx_{\mathrm{ctx}} v' : \mathtt{unit} \Leftrightarrow v = v' = ()$.
- ▷ *For names: For all $a, a' \in \mathbb{A}$,* $\vdash a \approx_{\mathrm{ctx}} a' : \mathtt{name} \Leftrightarrow a = a'$.
- ▷ *For data values: For $1 \leqslant k \leqslant K$,* $\vdash \mathtt{C}_k(v) \approx_{\mathrm{ctx}} \mathtt{C}_k(v') : \delta \Leftrightarrow \vdash v \approx_{\mathrm{ctx}} v' : \sigma_k$.

▷ *For pair values:* $\vdash (v_1,\ v_2) \approx_{\mathrm{ctx}} (v_1',\ v_2') : \tau_1 \times \tau_2$ *iff*

$$\vdash v_1 \approx_{\mathrm{ctx}} v_1' : \tau_1 \ \wedge \vdash v_2 \approx_{\mathrm{ctx}} v_2' : \tau_2.$$

▷ *For abstraction values: For all* $a, a' \in \mathbb{A}$ *and any* $a'' \in \mathbb{A} \setminus \mathrm{supp}(a, a', v, v')$*,*

$$\vdash \texttt{<<}a\texttt{>>}v \approx_{\mathrm{ctx}} \texttt{<<}a'\texttt{>>}v' : \texttt{<<name>>}\tau \Leftrightarrow \vdash (a\ a'') \cdot v \approx_{\mathrm{ctx}} (a'\ a'') \cdot v' : \tau.$$

▷ *For function values: For closed values* f*,* f'*,*

$$\vdash f \approx_{\mathrm{ctx}} f' : \tau \to \tau' \Leftrightarrow \forall v \in \mathrm{Val}_\tau.\ \vdash f\ v \approx_{\mathrm{ctx}} f'\ v : \tau'.$$

Proof. Observing the properties (5.2)–(5.6) required of the logical relation $\lhd_\tau^{\mathrm{val}}$ then the results follow by combining Lemma 5.5.6 with the definitions of $\mathcal{E}[\![\text{–}]\!]$ and $\mathcal{V}[\![\text{–}]\!]$. The most complicated case is that for function values, which we give here.

► *Case (fun).* Given $f \approx_{\mathrm{ctx}} f'$ then $f \prec_{\mathrm{ctx}} f'$ and $f' \prec_{\mathrm{ctx}} f$. Without loss of generality we may just consider the first case and show that $f \prec_{\mathrm{ctx}} f' \Leftrightarrow \forall v \in \mathrm{Val}_\tau.\ f\ v \prec_{\mathrm{ctx}} f'\ v$. By Lemma 5.5.6 it suffices to prove $f \leqslant_{\mathrm{v}} f' \Leftrightarrow \forall v \in \mathrm{Val}_\tau.\ f\ v \prec_{\mathrm{ctx}} f'\ v$. Observe that for closed values f of type $\tau \to \tau'$ and v of type τ, then $\mathcal{E}[\![f\ v]\!] = \mathcal{V}[\![f]\!]\mathcal{V}[\![v]\!]$. So if $f \leqslant_{\mathrm{v}} f'$ then $\mathcal{V}[\![f]\!] \lhd_{\tau\to\tau'}^{\mathrm{val}} f'$; hence for all $v \in \mathrm{Val}_\tau$ it follows (by the fundamental theorem and property (5.6) of the logical relation) that $\mathcal{V}[\![f]\!]\mathcal{V}[\![v]\!] \lhd_{\tau'}^{\mathrm{exp}} f'\ v$. Therefore $\mathcal{E}[\![f\ v]\!] \lhd_{\tau'}^{\mathrm{exp}} f'\ v$ and so $f\ v \leqslant_{\mathrm{e}} f'\ v$. It follows that $\vdash f\ v \prec_{\mathrm{ctx}} f'\ v : \tau'$; the reverse direction is similarly straightforward. □

Our final theorem and corollary in this section show what we can deduce when we know that two expressions, or canonical forms, have the same denotation.

Theorem 5.5.8 (Equality of denotation). *For expressions* e*,* e' *and a type* τ *such that* $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash e' : \tau$ *then*

$$\mathcal{E}[\![\Gamma \vdash e : \tau]\!] = \mathcal{E}[\![\Gamma \vdash e' : \tau]\!] \Rightarrow \Gamma \vdash e \approx_{\mathrm{ctx}} e' : \tau. \tag{5.73}$$

Proof. Assume $\mathcal{E}[\![\Gamma \vdash e : \tau]\!] = \mathcal{E}[\![\Gamma \vdash e' : \tau]\!]$. That is to say, for $\rho \in [\![\Gamma]\!]$ then $\mathcal{E}[\![\Gamma \vdash e : \tau]\!](\rho) = \mathcal{E}[\![\Gamma \vdash e' : \tau]\!](\rho)$. In particular given any closing substitution $\psi \in \mathrm{Subst}_\Gamma$, $\mathcal{E}[\![\Gamma \vdash e : \tau]\!]\mathcal{V}[\![\psi]\!] = \mathcal{E}[\![\Gamma \vdash e' : \tau]\!]\mathcal{V}[\![\psi]\!]$. Thus we can apply the computational adequacy result (Theorem 5.4.1) to deduce that the left-hand side of (5.73) implies $\langle S, e[\psi]\rangle\!\downarrow \Leftrightarrow \langle S, e'[\psi]\rangle\!\downarrow$ for all frame stacks S of argument type τ. Therefore $\Gamma \vdash e \approx_{\mathrm{ciu}} e' : \tau$, so Theorem 5.5.5 may be applied to deduce that $\Gamma \vdash e \approx_{\mathrm{ctx}} e' : \tau$. □

Corollary 5.5.9 (Denotation of closed canonical forms). *If* τ *and* δ *do not involve any occurrences of the function space constructor* $\to$ *(making them 'equality types' in the Standard ML terminology) and* v*,* v' *are closed values of type* τ*, then we have full abstraction:* $\mathcal{V}[\![v]\!] = \mathcal{V}[\![v']\!] \Leftrightarrow v \approx_{\mathrm{ctx}} v'$*. Thus* $\mathcal{E}[\![v]\!] = \mathcal{E}[\![v']\!] \Leftrightarrow v \approx_{\mathrm{ctx}} v'$*.*

Proof. The two bi-implications are clearly equivalent by virtue of Lemma 5.2.1. The forwards directions hold by virtue of Theorem 5.5.8. Considering the first, in the reverse direction we proceed by induction on the typing derivation of τ making use of the extensionality properties of Corollary 5.5.7. If $\tau = \texttt{unit}$ then v and v' must be `()`; thus $\mathcal{V}[\![v]\!] = \mathcal{V}[\![v']\!]$. Similarly, if $\tau = \texttt{name}$ then v and v' must both be the same atom and so we are done. If τ is the data type δ, then v and v' are of the form $\mathtt{C}_k(v_k)$ and $\mathtt{C}_k(v_k')$ respectively (for some $1 \leqslant k \leqslant K$), with $\mathcal{V}[\![v_k]\!] = \mathcal{V}[\![v_k']\!]$ by the induction hypothesis. It thus follows that $\mathcal{V}[\![v]\!] = \mathcal{V}[\![v']\!]$ by definition of $\mathcal{V}[\![\text{–}]\!]$. A similar argument applies at pair types. At abstraction types, where v and v' are of the form $\texttt{<<}a_1\texttt{>>}v_1$ and $\texttt{<<}a_2\texttt{>>}v_2$ respectively then the induction hypothesis tells us that $\mathcal{V}[\![(a_1\ a) \cdot v_1]\!] = \mathcal{V}[\![(a_2\ a) \cdot v_2]\!]$ for any $a \in \mathbb{A} \setminus \mathrm{supp}(a_1, a_2, v_1, v_2)$. Thus $\texttt{<<}a_1\texttt{>>}\mathcal{V}[\![v_1]\!] = \texttt{<<}a_2\texttt{>>}\mathcal{V}[\![v_2]\!]$, which by definition of $\mathcal{V}[\![\text{–}]\!]$ is $\mathcal{V}[\![\texttt{<<}a_1\texttt{>>}v_1]\!] = \mathcal{V}[\![\texttt{<<}a_2\texttt{>>}v_2]\!]$ as required. Finally, the function type case is excluded by the conditions we impose above. □

5.6 Algebraic identities

We are now in a position to use our denotational semantics to verify algebraic identities. For example, we can show that the semantics correctly handles 'garbage collection' of fresh names by proving

$$e \approx_{\mathrm{ctx}} \texttt{let}\ x\ \texttt{=}\ \texttt{fresh}\ \texttt{in}\ e$$

when x is not free in e. This is done as follows.

$$\begin{array}{lll}
& \mathcal{E}[\![\Gamma \vdash \texttt{let } x \texttt{ = fresh in } e : \tau]\!](\rho) & \\
= & \lambda\sigma \in [\![\tau]\!]^{\perp}.\, \mathcal{E}[\![\Gamma \vdash \texttt{fresh} : \texttt{name}]\!](\rho) & \\
& \quad (\underline{\lambda} a \in [\![\texttt{name}]\!].\, \mathcal{E}[\![\Gamma, x : \texttt{name} \vdash e : \tau]\!] & \text{by definition of } \mathcal{E}[\![\text{–}]\!] \\
& \quad (\rho[x \mapsto a])(\sigma) & \\
= & \lambda\sigma \in [\![\tau]\!]^{\perp}.\, \mathcal{E}[\![\Gamma, x : \texttt{name} \vdash e : \tau]\!](\rho[x \mapsto a])(\sigma) & \text{ditto, for } a \notin \mathrm{supp}(e, \sigma, \rho) \\
= & \lambda\sigma \in [\![\tau]\!]^{\perp}.\, \mathcal{E}[\![\Gamma \vdash e : \tau]\!](\rho)(\sigma) & \text{by Lemma 3.2.3, since } x \text{ not free in } e \\
= & \mathcal{E}[\![\Gamma \vdash e : \tau]\!](\rho) & \text{by } \eta\text{-reduction} \\
\Rightarrow & e \approx_{\mathrm{ctx}} \texttt{let } x \texttt{ = fresh in } e & \text{by Theorem 5.5.8.}
\end{array}$$

Whilst we do claim that our semantics is good for verifying certain correctness results of Mini-FreshML, we do not claim that it is the be-all-and-end-all. For there are certainly equivalences which we might wish to verify that are not provable in our current system. We take as an example the Mini-FreshML equivalent of an example presented by Stark[65, page 24, equation 12]. This considers the two terms

$$\texttt{let } n \texttt{ = fresh in fun } f(x) \texttt{ = if } x \texttt{ = } n \texttt{ then () else } \Omega$$

and $\texttt{fun } g(x) \texttt{ = } \Omega$, for a divergent term Ω of type `unit`. These two terms are indeed contextually equivalent, by virtue of the fact that the name (bound to n) encapsulated inside the support of the function f cannot be determined externally. However, if we attempt to prove the contextual equivalence of these two expressions via our denotational semantics then we get stuck. The problem is that we need to deduce that the following are equal:

$$\begin{array}{l}
\lambda\sigma \in [\![\texttt{name} \to \texttt{unit}]\!]^{\perp}.\, \sigma(\lambda d \in [\![\texttt{name}]\!].\, \lambda\sigma' \in [\![\texttt{unit}]\!]^{\perp}.\, \texttt{if } a, d, \sigma'(\top), \sigma'(\perp)) \\
\quad \text{(for some } a \notin \mathrm{supp}(\sigma)\text{); and} \\
\lambda\sigma \in [\![\texttt{name} \to \texttt{unit}]\!]^{\perp}.\, \sigma(\perp).
\end{array}$$

So Corollary 5.5.9 notwithstanding, we believe (as would be expected) that our semantics is in general not fully abstract. It seems likely that this is the case not only for the usual sequentiality reasons (à la PCF) but also due to the presence of the computational effects of generating fresh names. Unfortunately, examples such as the above do not appear to give a definite answer either way. However, recent work by Benton et al.[6] which uses a continuation-based model based on ours to reason about mutable store does support the view that the semantics is not fully abstract.

5.7 Correctness of representation

5.7.1 Background theory

In the previous sections we have established key properties of the denotational semantics and shown how these may be used to prove results concerning the operational behaviour of Mini-FreshML expressions. We now put this machinery into operation and turn to the issue of proving the correctness results from §3.7.

Recall the use of the untyped λ-calculus from that Section as an example object language. In order to model λ-terms we can simply set $K = 3$, $\sigma_1 = \texttt{name}$, $\sigma_2 = \texttt{<<name>>}\delta$ and $\sigma_3 = \delta \times \delta$. Then writing `Var` to stand for $\mathtt{C}_1$, `Lam` to stand for $\mathtt{C}_2$ and `App` to stand for $\mathtt{C}_3$, the type δ corresponds to the Fresh O'Caml declaration

$$\texttt{type } \delta \texttt{ = Var of name | Lam of <<name>>}\delta \texttt{ | App of } \delta \texttt{ * } \delta.$$

The FM-cppo $[\![\delta]\!]$ is the minimal invariant solution $\mathcal{D}$ to the recursive domain equation $i : F(\mathcal{D}) \cong \mathcal{D}$, where

$$F(\mathcal{D}) \stackrel{\mathrm{def}}{=} \mathbb{A} \oplus ([\mathbb{A}]D) \oplus (\mathcal{D} \otimes \mathcal{D}). \tag{5.74}$$

Note that the description in §4.5 provides for a more general method of solution than is required here, due to the lack of contravariant occurrences of the variable $\mathcal{D}$.

Definition 5.7.1 (Translation to values). For each λ-term t, define a Mini-FreshML value $[t]_{\mathrm{v}} \in \mathrm{Val}_{\delta}$ by induction on the structure of t as follows.

$$[x]_{\mathrm{v}} \stackrel{\mathrm{def}}{=} \texttt{Var}(x)$$

$$\begin{array}{rcl}[\lambda x.\ t]_{\mathrm{v}} & \stackrel{\mathrm{def}}{=} & \mathtt{Lam(<<}x\mathtt{>>}[t]_{\mathrm{v}}\mathtt{)} \\ [t\ t']_{\mathrm{v}} & \stackrel{\mathrm{def}}{=} & \mathtt{App(}[t]_{\mathrm{v}}\mathtt{,}\ [t']_{\mathrm{v}}\mathtt{).}\end{array}$$

Thus if t is a closed (resp. open) term, $[t]_{\mathrm{v}}$ is a closed (resp. open) value. $\diamond$

Lemma 5.7.2. $[-]_{\mathrm{v}}$ *is equivariant: if* $x, x' \in \mathrm{VId}$ *then* $(x\ x') \cdot [t]_{\mathrm{v}} = [(x\ x') \cdot t]_{\mathrm{v}}$. $\square$

Lemma 5.7.3. *For all* λ*-terms* t, t' *whose variables (free* and *bound) are contained within a set of identifiers* $\overline{x} = \{x_1, \cdots, x_n\}$ *and an injective value substitution* $\psi : \mathrm{VId} \to \mathbb{A}$ *with* $\mathrm{dom}(\psi) \subseteq \overline{x}$*,*

$$[t]_{\mathrm{v}}[\psi] \approx_{\mathrm{ctx}} [t']_{\mathrm{v}}[\psi] \Leftrightarrow t \equiv_\alpha t'.$$

Proof. By induction on the structure of t. Note that at each stage we can assume that the structure of t' is the same as t, for in the forwards direction this follows from the extensionality properties of Lemma 5.5.7 and in the reverse direction this follows from the axiom and rules inductively defining the relation $\equiv_\alpha$ (Definition 3.7.1).

► *Case (var).* For variables x, x' (which must lie in the domain of ψ) then we wish to know

$$\mathtt{Var(}\psi(x)\mathtt{)} \approx_{\mathrm{ctx}} \mathtt{Var(}\psi(x')\mathtt{)} \Leftrightarrow x \equiv_\alpha x'.$$

In the forwards direction, the extensionality properties of Corollary 5.5.7 tell us that $\psi(x)$ and $\psi(x')$ must be the same atom. Since ψ is injective then $x = x'$ and therefore $x \equiv_\alpha x'$. In the reverse direction, we must have $x = x'$ and conclude by reflexivity of $\approx_{\mathrm{ctx}}$.

► *Case (abst).* We wish to know that

$$\mathtt{Lam(<<}\psi(x)\mathtt{>>(}[t]_{\mathrm{v}}[\psi]\mathtt{))} \approx_{\mathrm{ctx}} \mathtt{Lam(<<}\psi(x')\mathtt{>>(}[t]_{\mathrm{v}}[\psi]\mathtt{))} \Leftrightarrow \lambda x.\ t \equiv_\alpha \lambda x'.\ t'$$

for terms t, t' and variables x, x' in the domain of ψ. The extensionality properties tell us that the left-hand side holds iff $\mathtt{<<}\psi(x)\mathtt{>>(}[t]_{\mathrm{v}}[\psi]\mathtt{)} \approx_{\mathrm{ctx}} \mathtt{<<}\psi(x')\mathtt{>>(}[t']_{\mathrm{v}}[\psi]\mathtt{)}$ and therefore just when

$$(\psi(x)\ a) \cdot ([t]_{\mathrm{v}}[\psi]) \approx_{\mathrm{ctx}} (\psi(x')\ a) \cdot ([t']_{\mathrm{v}}[\psi]) \tag{5.75}$$

for some/any $a \in \mathbb{A} \setminus \mathrm{supp}(\psi(x), \psi(x'), [t]_{\mathrm{v}}[\psi], [t']_{\mathrm{v}}[\psi])$. By the induction hypothesis we have that $[t]_{\mathrm{v}}[\psi] \approx_{\mathrm{ctx}} [t']_{\mathrm{v}}[\psi] \Leftrightarrow t \equiv_\alpha t'$, since all the variables occurring in t and t' are mapped by ψ. Moreover, we also have $[t]_{\mathrm{v}}[\psi'] \approx_{\mathrm{ctx}} [t']_{\mathrm{v}}[\psi'] \Leftrightarrow t \equiv_\alpha t'$, where ψ' is that substitution ψ' mapping a fresh variable x'' to a and acting as ψ otherwise. Since swapping does not affect the size of the subterms t and t' we can therefore deduce that

$$[(x\ x'') \cdot t]_{\mathrm{v}}[\psi'] \approx_{\mathrm{ctx}} [(x'\ x'') \cdot t']_{\mathrm{v}}[\psi'] \Leftrightarrow (x\ x'') \cdot t \equiv_\alpha (x'\ x'') \cdot t'.$$

Lemma 5.7.2 now tells us that $(x\ x'') \cdot t \equiv_\alpha (x'\ x'') \cdot t'$ iff $(\psi'(x)\ \psi'(x'')) \cdot ([t]_{\mathrm{v}}[\psi']) \approx_{\mathrm{ctx}} (\psi'(x')\ \psi'(x'')) \cdot ([t']_{\mathrm{v}}[\psi'])$. Now observe that: $\psi'(x) = \psi(x)$ (and similarly for x'); and $[t]_{\mathrm{v}}[\psi'] = [t]_{\mathrm{v}}[\psi]$ (similarly for t') since x'' is fresh for both t and t'. These facts together with the rule (3.8) enable us to deduce

$$(\psi(x)\ a) \cdot ([t]_{\mathrm{v}}[\psi]) \approx_{\mathrm{ctx}} (\psi(x')\ a) \cdot ([t']_{\mathrm{v}}[\psi]) \Leftrightarrow \lambda x.\ t \equiv_\alpha \lambda x'.\ t'.$$

However, the left-hand side of this holds by (5.75).

► *Case (app).* We wish to deduce that

$$[t_1\ t_2]_{\mathrm{v}}[\psi] \approx_{\mathrm{ctx}} [t_1'\ t_2']_{\mathrm{v}}[\psi] \Leftrightarrow t_1\ t_2 \equiv_\alpha t_1'\ t_2' \tag{5.76}$$

using the induction hypotheses: $[t_1]_{\mathrm{v}}[\psi] \approx_{\mathrm{ctx}} [t_1]_{\mathrm{v}}[\psi] \Leftrightarrow t_1 \equiv_\alpha t_1'$ (and similarly for t_2, t_2'). However, using the extensionality properties and the rule (3.9) it is immediate that the conjunction of these two statements holds just when (5.76) does also. $\square$

In order to derive correctness results pertaining to expressions which are not in canonical form we shall adopt a method due to Pitts (unpublished manuscript). This identifies λ-terms in a certain way so as to make distinct and disjoint the free and bound variables of each term—effectively enforcing the Barendregt variable convention. For disjoint subsets $\overline{x}, \overline{x}'$ of VId, define the subset $\Lambda(\overline{x}; \overline{x}') \subseteq \Lambda$ inductively by the following rules:

$$\frac{x \in \overline{x}}{x \in \Lambda(\overline{x}, \emptyset)} \qquad \frac{t \in \Lambda(\{x\} \uplus \overline{x}, \overline{x}')}{\lambda x.\ t \in \Lambda(\overline{x}, \{x\} \cup \overline{x}')}$$

$$\frac{t \in \Lambda(\overline{x}, \overline{x}'_1) \qquad t' \in \Lambda(\overline{x}, \overline{x}'_2) \qquad \overline{x}'_1 \cap \overline{x}'_2 = \emptyset}{t\ t' \in \Lambda(\overline{x}, \overline{x}'_1 \cup \overline{x}'_2)}.$$

If $t \in \Lambda(\overline{x}, \overline{x}')$ then: the free variables of t are contained within $\overline{x}$; the bound variables of t are mutually distinct and are contained within $\overline{x}'$; the sets of free and bound variables of t are disjoint; and $\mathrm{vars}(t) \subseteq \overline{x} \cup \overline{x}'$. Each term $t \in \Lambda$ is α-equivalent to a term in $\Lambda(\overline{x}, \overline{x}')$ for some $\overline{x}$, $\overline{x}'$.

Lemma 5.7.4. *For $t \in \Lambda(\overline{x}, \overline{x}')$ and an injective substitution $\psi : \mathrm{VId} \to \mathbb{A}$ with $\mathrm{dom}(\psi) \subseteq \overline{x} \cup \overline{x}'$ then $[t]_{\mathrm{e}}[\psi] \approx_{\mathrm{ctx}} [t]_{\mathrm{v}}[\psi]$: $\mathtt{lam}$.*

Proof. Either a 'denotational' or 'operational'[11] approach may be taken; we choose the former. By Theorem 5.5.8 it suffices to show that for all $\sigma \in [\![\mathtt{lam}]\!]^{\perp}$,

$$\mathcal{E}[\![\vdash [t]_{\mathrm{e}}[\psi] : \mathtt{lam}]\!](\emptyset)(\sigma) = \sigma(\mathcal{V}[\![\vdash [t]_{\mathrm{v}}[\psi] : \mathtt{lam}]\!](\emptyset)).$$

That this is so follows by induction on the derivation of $t \in \Lambda(\overline{x}, \overline{x}')$.

► *Case (var).* Follows immediately from Lemma 5.2.1, since $[x]_{\mathrm{e}}$ is a canonical form.

► *Case (abst).* Under the assumption that $\mathcal{E}[\![[t]_{\mathrm{e}}[\psi]]\!](\sigma) = \sigma(\mathcal{V}[\![[t]_{\mathrm{v}}[\psi]]\!])$, where $t \in \Lambda(\{x\} \uplus \overline{x}, \overline{x}')$ and $\mathrm{dom}(\psi) \subseteq \{x\} \uplus \overline{x} \uplus \overline{x}'$, we wish to show that

$$\mathcal{E}[\![[\lambda x.\ t]_{\mathrm{e}}[\psi]]\!](\sigma) = \sigma(\mathcal{V}[\![[\lambda x.\ t]_{\mathrm{v}}[\psi]]\!]) \tag{5.77}$$

where $\lambda x.\ t \in \Lambda(\overline{x}, \overline{x}' \uplus \{x\})$. Completely expanding the definition of $\mathcal{E}[\![\text{–}]\!]$ on the left-hand side we obtain

$$\mathcal{E}[\![[t]_{\mathrm{e}}[\psi|_{\overline{x} \uplus \overline{x}'}]]\!]\{x \mapsto a'\}(\lambda d \in [\![\mathtt{lam}]\!].\ \sigma((i \circ \mathrm{in}_2)[a']d)) \tag{5.78}$$

for some $a' \in \mathbb{A} \setminus \mathrm{supp}([t]_{\mathrm{v}}[\psi], [t]_{\mathrm{e}}[\psi], \psi(x), \sigma)$ and where $\psi|_{\overline{x} \uplus \overline{x}'}$ is the restriction of ψ to map only from $\overline{x}$ and $\overline{x}'$. Observing that we have $x \notin \overline{x}$ and $x \notin \overline{x}'$, extend $\psi|_{\overline{x} \uplus \overline{x}'}$ to form a new substitution ψ' behaving as $\psi|_{\overline{x} \uplus \overline{x}'}$ but also mapping x to a'. This ψ' is also injective and therefore we can make use of the fact (Lemma 5.3.4) that $\mathcal{E}[\![e]\!]\{x \mapsto a'\} = \mathcal{E}[\![e[a'/x]]\!](\emptyset)$ for any expression e with a single free value identifier x of type $\mathtt{name}$ to deduce that (5.78) is equivalent to

$$\mathcal{E}[\![[t]_{\mathrm{e}}[\psi']]\!](\lambda d \in [\![\mathtt{lam}]\!].\ \sigma((i \circ \mathrm{in}_2)[a']d))$$

which, via an application of the induction hypothesis contracts to

$$\sigma((i \circ \mathrm{in}_2)[a']\mathcal{V}[\![[t]_{\mathrm{v}}[\psi']]\!]). \tag{5.79}$$

Write a for the atom $\psi(x)$. It remains to show that (5.79) is equal to $\sigma((i \circ \mathrm{in}_2)[a]\mathcal{V}[\![[t]_{\mathrm{v}}[\psi]]\!])$, at which point we can apply the definition of $\mathcal{V}[\![\text{–}]\!]$ to conclude (5.77). By Lemma 4.2.24, since $a \neq a'$ and $a \notin \mathrm{supp}([t]_{\mathrm{v}}[\psi])$ then it suffices to show that $(a\ a') \cdot \mathcal{V}[\![[t]_{\mathrm{v}}[\psi]]\!] = \mathcal{V}[\![[t]_{\mathrm{v}}[\psi']]\!]$. The left-hand side of this is equal to $\mathcal{V}[\![(a\ a') \cdot ([t]_{\mathrm{v}}[\psi])]\!]$ by Lemma 5.3.1 and then we are done, since it is easy to see that $(a\ a') \cdot ([t]_{\mathrm{v}}[\psi]) = [t]_{\mathrm{v}}[\psi']$.

► *Case (app).* We take as assumptions that $\mathcal{E}[\![[t]_{\mathrm{e}}[\psi_1]]\!](\sigma) = \sigma(\mathcal{V}[\![[t]_{\mathrm{v}}[\psi_1]]\!])$ and $\mathcal{E}[\![[t']_{\mathrm{e}}[\psi_2]]\!](\sigma) = \sigma(\mathcal{V}[\![[t']_{\mathrm{v}}[\psi_2]]\!])$ where $t \in \Lambda(\overline{x}, \overline{x}'_1)$, $t' \in \Lambda(\overline{x}, \overline{x}'_2)$ and $\overline{x}'_1 \cap \overline{x}'_2 = \emptyset$. Let us take the substitutions ψ_1, ψ_2 to be $\psi|_{\overline{x} \cup \overline{x}'_1}$ and $\psi|_{\overline{x} \cup \overline{x}'_2}$ respectively (these are still injective since ψ is). Carefully observing the domains of these substitutions we see that we need to prove

$$\mathcal{E}[\![[t\ t']_{\mathrm{e}}[\psi]]\!](\sigma) = \sigma(\mathcal{V}[\![[t\ t']_{\mathrm{v}}[\psi]]\!]) = \sigma(\mathcal{V}[\![\mathtt{App}([t]_{\mathrm{v}}[\psi_1],\ \mathcal{V}[\![[t']_{\mathrm{v}}[\psi_2])]\!]]\!]).$$

We proceed as in the previous case by fully expanding the left-hand side to obtain

$$\mathcal{E}[\![[t]_{\mathrm{e}}[\psi_1]]\!](\lambda d \in [\![\mathtt{lam}]\!].\ \mathcal{E}[\![[t']_{\mathrm{e}}[\psi_2]]\!](\lambda d' \in [\![\mathtt{lam}]\!].\ \sigma((i \circ \mathrm{in}_3)\langle d, d'\rangle)))$$

which from the assumptions is $\sigma((i \circ \mathrm{in}_3)(\mathcal{V}[\![[t]_{\mathrm{v}}[\psi_1]]\!], \mathcal{V}[\![[t']_{\mathrm{v}}[\psi_2]]\!]))$. □

Lemma 5.7.5 (Divergent terms). *For a typeable closed expression e of type $\mathtt{lam}$ and the divergent term $\Omega \stackrel{\mathrm{def}}{=} \mathtt{(fun\ f(x) = f(x))(())}$ then*

$$e \approx_{\mathrm{ctx}} \Omega \Leftrightarrow \forall S : \mathtt{lam} \multimap \_.\ \mathcal{E}[\![e]\!]\mathcal{S}[\![S]\!] = \perp \Leftrightarrow \forall S : \mathtt{lam} \multimap \_.\ \langle S, e\rangle\!\downarrow$$

where we write $\langle S, e\rangle\!\downarrow$ to indicate that $\langle S, e\rangle\!\downarrow$ does not hold.

[11] Such a proof argues along the same lines, but mainly by making observations relating to the termination relation. Morally speaking, we feel that the 'denotational' approach is more in line with the style of semantics presented here.

Proof. $e \approx_{\mathrm{ctx}} \Omega$ holds iff $e \approx_{\mathrm{ciu}} \Omega$, i.e. for all S of type $\mathtt{lam} \multimap \_$, $\langle S, e\rangle\!\downarrow$ holds just in case $\langle S, \Omega\rangle\!\downarrow$. From the definition of the termination relation, we have that $\langle S, \Omega\rangle\!\downarrow$ never holds; thus, $\langle S, e\rangle\!\not\downarrow$. Combining this with the computational adequacy result (Theorem 5.4.1) concludes the proof. □

5.7.2 Correctness results

Theorem 5.7.6 (Correctness for values). *The contextual equivalence classes of closed values of type* `lam` *are in bijection with the α-equivalence classes of λ-terms with variables in* VId.

Proof. Fix any bijection[12] $\psi : \mathrm{VId} \cong \mathbb{A}$. Then given any two α-equivalent terms t, t', Lemma 5.7.3 tells us that $[t]_{\mathrm{v}}[\psi] \approx_{\mathrm{ctx}} [t']_{\mathrm{v}}[\psi]$. In the reverse direction, we need to show that any two contextually-equivalent values v, v' of type `lam` arise from α-equivalent terms t, t' such that $v = [t]_{\mathrm{v}}[\psi]$ and $v' = [t']_{\mathrm{v}}[\psi]$, for then we can apply the same Lemma in the other direction to get the desired result. However, given any value v of type `lam` then the typing rules tell us that it can only be of the form `Var(`a`)` for some atom a, `Lam(<<`a`>>`v'`)` for some value v' of type `lam`, or `App(`v_1`,` v_2`)` for some values v_1 and v_2 of type `lam`. It is thus straightforward to see that every value of type `lam` corresponds to a value $[t]_{\mathrm{v}}[\psi]$. □

Corollary 5.7.7 (Equality of denotation at type `lam`). *Given any two λ-terms t, t' with variables in* VId *and an injective substitution $\psi : \mathrm{VId} \to \mathbb{A}$ then*

$$\mathcal{V}[\![[t]_{\mathrm{v}}[\psi]]\!] = \mathcal{V}[\![[t]_{\mathrm{v}}[\psi]]\!] \Leftrightarrow t \equiv_\alpha t'.$$

Proof. Follows immediately from Corollary 5.5.9 and Theorem 5.7.6. □

The previous theorem tells us something about the relation between elements of $[\![\mathtt{lam}]\!]$ and terms of the λ-calculus. What it does not however tell us is that the $v \mapsto \mathcal{V}[\![v]\!]$ map is surjective onto $[\![\mathtt{lam}]\!]_\downarrow$. The following Theorem establishes this useful 'no junk' property.

Theorem 5.7.8 (No junk in $[\![\mathtt{lam}]\!]$). *The non-bottom elements of $[\![\mathtt{lam}]\!]$ are in bijection with the α-equivalence classes of λ-terms with variables in* VId.

Proof. Pitts and Gabbay establish in previous work[20] that the elements of the inductively-defined FM-set $\Lambda/{\equiv_\alpha} \stackrel{\mathrm{def}}{=} \mu X.\ \mathbb{A} + ([\mathbb{A}]X) + (X \times X)$ are in bijection with the α-equivalence classes of λ-terms with variables in $\mathbb{A}$. Since such λ-terms are in bijection with those containing variables in VId, it suffices to establish that the lifted FM-set $(\Lambda/{\equiv_\alpha})_\perp$ is in bijection with $[\![\mathtt{lam}]\!]$, provided that the least elements of these are mapped to each other. This may be done neatly by exhibiting an isomorphism $j : F((\Lambda/{\equiv_\alpha})_\perp, (\Lambda/{\equiv_\alpha})_\perp) \cong (\Lambda/{\equiv_\alpha})_\perp$ such that the pair $((\Lambda/{\equiv_\alpha})_\perp, j)$ has the minimal invariant property for the functor F of §5.2.1. Since such pairs are unique up to isomorphism (by Theorem 4.5.2), we can deduce that $[\![\mathtt{lam}]\!]$ is isomorphic to $(\Lambda/{\equiv_\alpha})_\perp$ so long as the least elements are correctly mapped. We must therefore exhibit some j with $j(\perp) = \perp$, $j^{-1}(\perp) = \perp$ and $\mathrm{fix}(\Phi) = id_{(\Lambda/{\equiv_\alpha})_\perp}$, where $\Phi \stackrel{\mathrm{def}}{=} \lambda f \in (\Lambda/{\equiv_\alpha})_\perp \multimap (\Lambda/{\equiv_\alpha})_\perp.\ j \circ F(f, f) \circ j^{-1}$. A suitable equivariant j is as follows:

$$j(\perp) \stackrel{\mathrm{def}}{=} \perp \quad j(\mathrm{in}_1(a \in \mathbb{A})) \stackrel{\mathrm{def}}{=} a \quad j(\mathrm{in}_2([a]t)) \stackrel{\mathrm{def}}{=} \lambda a.\ t \quad j(\mathrm{in}_3(t, t')) \stackrel{\mathrm{def}}{=} t\ t'$$

where the functions in_k are the usual injections into a coalesced sum. Observe that since $\Phi^{n+1}(\perp)(t) \stackrel{\mathrm{def}}{=} \Phi(\Phi^n(\perp))(t)$ then

$$\Phi^{n+1}(\perp)(t) = \begin{cases} \perp & \text{if } t = \perp; \\ a & \text{if } t = a \in \mathbb{A}; \\ \lambda a'.\ (\Phi^n)(\perp)((a\ a') \cdot t') & \text{if } t = \lambda a.\ t', \text{ with} \\ & a' \in \mathbb{A} \setminus \mathrm{supp}(\Phi^n(\perp), a, t'); \\ (\Phi^n(\perp)(t_1))\ (\Phi^n(\perp)(t_2)) & \text{if } t = t_1\ t_2. \end{cases}$$

We are first going to show that $\Phi^n(\perp)$ has empty support for any n. Proceed by induction on n. For the base case, $\Phi^0(\perp)(t) \stackrel{\mathrm{def}}{=} \perp$ which has empty support. Then under the assumption

[12] If working formally inside FM-set theory, we would have to do this differently since this construction is not finitely-supported.

that $\pi \cdot (\Phi^n(\bot)(t)) = \Phi^n(\bot)(\pi \cdot t)$ we wish to know $\pi \cdot (\Phi^{n+1}(\bot)(t)) = \Phi^{n+1}(\bot)(\pi \cdot t)$. This is straightforward to see in all cases except where t is a λ-abstraction, say $\lambda a.\ t'$. Then for $a' \in \mathbb{A} \setminus \mathrm{supp}(\Phi^n(\bot), a, t')$ we have

$$\begin{array}{lll}
\pi \cdot (\Phi^{n+1}(\bot)(t)) & & \\
\quad = & \pi \cdot \lambda a'.\ \Phi^n(\bot)((a\ a') \cdot t') & \text{by definition} \\
\quad = & \lambda \pi(a').\ \pi \cdot (\Phi^n(\bot)((a\ a') \cdot t')) & \text{swapping on the syntax tree} \\
\quad = & \lambda \pi(a').\ \Phi^n(\bot)((\pi(a)\ \pi(a')) \cdot \pi \cdot t') & \text{by the induction hypothesis} \\
\quad = & \lambda a''.\ \Phi^n(\bot)((a''\ \pi(a')) \cdot \pi \cdot t') & \text{writing } a'' = \pi(a) \\
\quad = & \Phi^{n+1}(\bot)(\pi \cdot t') & \text{since } \pi(a') \notin \mathrm{supp}(\pi \cdot t').
\end{array}$$

Given this equivariance result the previous definition may be simplified to

$$\Phi^{n+1}(\bot)(t) = \begin{cases} \bot & \text{if } t = \bot; \\ a & \text{if } t = a \in \mathbb{A}; \\ \lambda a.\ (\Phi^n)(\bot)(t') & \text{if } t = \lambda a.\ t'; \\ (\Phi^n(\bot)(t_1))\ (\Phi^n(\bot)(t_2)) & \text{if } t = t_1\ t_2. \end{cases}$$

We can now see by induction on n that $\Phi^n(\bot)$ is the identity on terms of height at most n (where $\bot$ is the only term of height zero). Therefore $\mathrm{fix}(\Phi) = id_{(\Lambda/\equiv_\alpha)_\bot}$. $\square$

Theorem 5.7.9 (Correctness for expressions). *For λ-terms t, t' with free variables contained in the finite set $\{x_0, \cdots, x_n\} \subseteq \mathrm{VId}$,*

$$t \equiv_\alpha t \Leftrightarrow \{x_0\ \texttt{:}\ \texttt{name}, \cdots, x_n\ \texttt{:}\ \texttt{name}\} \vdash [t]_\mathrm{e} \approx_\mathrm{ctx} [t']_\mathrm{e}\ \texttt{:}\ \texttt{lam}.$$

Proof. For the forwards direction we proceed by induction on the derivation of $t \equiv_\alpha t'$ to show that

$$t \equiv_\alpha t' \Rightarrow [t]_\mathrm{e} = [t']_\mathrm{e}, \tag{5.80}$$

where $- = -$ is equality of Mini-FreshML expressions, identified up to α-conversion of bound value identifiers.

▶ *Case (var).* The terms t and t' must both be the same value identifier $x \in \mathrm{VId}$ and so we are done.

▶ *Case (lam).* We have that $\lambda x.\ t \equiv_\alpha \lambda x'.\ t'$ and wish to know that

$$\begin{array}{l} \texttt{let}\ x\ \texttt{=}\ \texttt{fresh in Lam(<<}x\texttt{>>}[t]_\mathrm{e}) = \\ \texttt{let}\ x'\ \texttt{=}\ \texttt{fresh in Lam(<<}x'\texttt{>>}[t']_\mathrm{e}). \end{array} \tag{5.81}$$

The $\equiv_\alpha$ judgement must have been derived by knowing that $(x\ x'') \cdot t \equiv_\alpha (x'\ x'') \cdot t'$ for some $x'' \in \mathbb{A} \setminus \mathrm{supp}(t, t')$. Therefore, Lemma 3.7.3 and the induction hypothesis give us that $(x\ x'') \cdot [t]_\mathrm{e} = (x'\ x'') \cdot [t']_\mathrm{e}$ and hence

$$\begin{array}{l} \texttt{let}\ x''\ \texttt{=}\ \texttt{fresh in Lam(<<}x''\texttt{>>}(x\ x'') \cdot [t]_\mathrm{e}) = \\ \texttt{let}\ x''\ \texttt{=}\ \texttt{fresh in Lam(<<}x''\texttt{>>}(x'\ x'') \cdot [t']_\mathrm{e}). \end{array}$$

However since x'' does not occur in t or t' (and therefore not in $[t]_\mathrm{e}$ or $[t']_\mathrm{e}$) then the expression $\texttt{let}\ x''\ \texttt{=}\ \texttt{fresh in Lam(<<}x''\texttt{>>}(x\ x'') \cdot [t]_\mathrm{e})$ is in the same α-equivalence class as $\texttt{let}\ x\ \texttt{=}\ \texttt{fresh in Lam(<<}x\texttt{>>}[t]_\mathrm{e})$. A similar argument applies for the t' side of the equality. We thus have (5.81) as required.

▶ *Case (app).* We have that $t_1\ t_2 \equiv_\alpha t_1'\ t_2'$ and want $\texttt{App}([t_1]_\mathrm{e},\ [t_2]_\mathrm{e}) = \texttt{App}([t_1']_\mathrm{e},\ [t_2']_\mathrm{e})$. However the induction hypotheses tell us that $t_1 \equiv_\alpha t_1' \Rightarrow [t_1]_\mathrm{e} = [t_1']_\mathrm{e}$ and similarly for t_2, t_2'.

For the reverse direction of the theorem, by renaming variables we can find a finite set $\overline{x}'$ and terms $t_\Lambda, t'_\Lambda \in \Lambda(\overline{x}, \overline{x}')$ such that $t_\Lambda \equiv_\alpha t$ and $t'_\Lambda \equiv_\alpha t'$. Applying (5.80) then yields that $[t_\Lambda]_\mathrm{e} \approx_\mathrm{ctx} [t'_\Lambda]_\mathrm{e}\ \texttt{:}\ \texttt{name}$. Choosing some injective substitution $\psi\ \texttt{:}\ \mathrm{VId} \to \mathbb{A}$ with domain $\overline{x} \cup \overline{x}'$, we can apply Lemma 5.7.4 to conclude that $[t_\Lambda]_\mathrm{v}[\psi] \approx_\mathrm{ctx} [t'_\Lambda]_\mathrm{v}[\psi]\ \texttt{:}\ \texttt{lam}$. Finally, we apply Lemma 5.7.3 to obtain $t \equiv_\alpha t_\Lambda \equiv_\alpha t'_\Lambda \equiv_\alpha t'$. $\square$

Theorem 5.7.10 (Form of expressions). *For a closed Mini-FreshML expression e of type* `lam` *and a divergent term Ω, then either $e \approx_\mathrm{ctx} \Omega$ or there exists a closed value v of type* `lam` *with*

$$e \approx_\mathrm{ctx} \texttt{let}\ x_1\ \texttt{=}\ \texttt{fresh in}\ \cdots\ \texttt{let}\ x_n\ \texttt{=}\ \texttt{fresh in}\ v,$$

for value identifiers $\{x_1, \ldots, x_n\}$.

Proof. Using Lemma 5.7.5 we see that if $\vdash e \approx_{\mathrm{ctx}} \Omega$ does *not* hold, then there exists some frame stack S such that $\langle S, e\rangle\downarrow$. Therefore by Lemma 3.4.3 we know that $\langle [], e\rangle\downarrow$. We can now apply the forwards direction of Theorem 3.4.6 to deduce that there exists some closed value v' of type `lam` and finite sets of atoms $\overline{a}' \supseteq \overline{a} \supseteq \mathrm{atms}(e)$ such that: $\overline{a}, e \Downarrow v', \overline{a}'$ with $\mathrm{atms}(v') \subseteq \overline{a}'$ and $\langle [], v'\rangle\downarrow$. Pick a finite set $X \subseteq \mathrm{VId}$ with cardinality $|\overline{a}' \setminus \overline{a}|$ and let ψ be a bijection $X \cong (\overline{a}' \setminus \overline{a})$. Then we can write v' as $v[\psi]$, where v is that value formed from v' by replacing each atom $a \in \overline{a}' \setminus \overline{a}$ which occurs in v' with a value identifier $\psi^{-1}(a)$. Enumerating X as $\{x_1, \ldots, x_n\}$ we now show that

$$e \approx_{\mathrm{ciu}} \texttt{let } x_1 \texttt{ = fresh in } \cdots \texttt{ let } x_n \texttt{ = fresh in } v,$$

so we can conclude by Theorem 5.5.5. Call the right-hand side of this conjectured equivalence e'.

In the forwards direction, assume $\langle S, e\rangle\downarrow$. Theorem 3.4.6 then tells us that there exist sets of atoms $\overline{a}'_1 \supseteq \overline{a}_1 \supseteq \mathrm{atms}(S, e)$ such that: $\overline{a}_1, e \Downarrow v'_1, \overline{a}'_1$ with $\mathrm{atms}(v'_1) \subseteq \overline{a}'_1$ and $\langle S, v'_1\rangle\downarrow$. Then by Lemma 3.3.3 there exists a bijection[13] $\pi : (\overline{a}'_1 \setminus \overline{a}_1) \cong (\overline{a}' \setminus \overline{a})$ with $\pi \cdot v'_1 = v'$. Since π^{-1} fixes $\mathrm{atms}(e)$ pointwise then $\langle S, v[\pi^{-1} \cdot \psi]\rangle\downarrow$, where $\pi^{-1} \cdot \psi$ maps each $x \in \mathrm{dom}(\psi)$ to $\pi^{-1}(\psi(x))$. It now follows that

$$e \prec_{\mathrm{ciu}} \texttt{let } x_1 \texttt{ = } \pi^{-1}(\psi(x_1)) \texttt{ in } \cdots \texttt{ let } x_n \texttt{ = } \pi^{-1}(\psi(x_n)) \texttt{ in } v.$$

However for each $1 \leqslant i \leqslant n$, $a = \pi^{-1}(\psi(x_i)) \in (\overline{a}'_1 \setminus \overline{a}_1)$ and therefore $a \notin \mathrm{atms}(S)$. We therefore have $e \prec_{\mathrm{ciu}} e'$.

In the reverse direction, assume $\langle S, e'\rangle\downarrow$. It suffices to show $\langle S, v[\psi]\rangle\downarrow$, since then we can conclude by Theorem 3.4.6 (for we already have that $\overline{a}, e \Downarrow v[\psi], \overline{a}'$ from above). Pick a finite set of atoms $\overline{a}_2 \supseteq \mathrm{supp}(S, e)$ together with n distinct atoms $\{a_1, \ldots, a_n\}$; let $\overline{a}'_2 \stackrel{\mathrm{def}}{=} \overline{a}_2 \cup \{a_1, \ldots, a_n\}$. Fix a bijection $\pi : (\overline{a}' \setminus \overline{a}) \cong (\overline{a}'_2 \setminus \overline{a}_2)$ and observe that $\langle S, e'\rangle\downarrow$ implies $\langle S, v[\pi \cdot \psi]\rangle\downarrow$ (since each atom in $\overline{a}'_2 \setminus \overline{a}_2$ is not in the support of S). Since π fixes all atoms in v, we know that $\langle S, \pi \cdot (v[\psi])\rangle\downarrow$; it follows by the equivariance property of the termination relation (and indeed that π^{-1} fixes the support of S pointwise) that $\langle S, v[\psi]\rangle\downarrow$ as required. □

This final theorem concludes our work on the denotational semantics of Mini-FreshML. We have shown how a novel monadic semantics can provide a straightforward setting for modelling dynamic allocation. Using the semantics, we have shown how object language terms may be translated into Mini-FreshML expressions and proved how such expressions correspond to the α-equivalence classes of the original terms. Such formal proof, long though it may be, is technically satisfying and provides good evidence that our syntax-manipulating paradigm is sound.

In the next chapter then, we shall exchange the art of mathematics for the art of programming. We will discuss the innards of the Fresh O'Caml compiler together with some of the past and future issues surrounding the implementation.

[13] Thought of as a permutation, which fixes all atoms not in its domain.

6 Implementation

'Theory attracts practice as the magnet attracts iron.' —Gauss

IN THIS CHAPTER we describe the implementation of the freshness features in Fresh O'Caml. The text here corresponds to the current version of the implementation[60, 62], but since this is still in its infancy we also discuss technologies which are not yet incorporated into that system.

6.1 Library, or bespoke system?

At the start of Fresh O'Caml's development, it was necessary to decide how the new features would be integrated into the existing language system. In such a scenario there are two basic options[1]:

- implementation as a standard library module, leaving the main compiler and runtime systems unchanged;
- wholesale extension of the compiler and runtime systems themselves.

The first of these corresponds to an embedding of the freshness features within the target language whilst the second corresponds to a fundamental extension of the language. Fresh O'Caml falls into the second category: a decision which was reached after consideration of the following issues.

► *The necessity of swapping.* Theoretically speaking, Fresh O'Caml code could be source-to-source translated into standard O'Caml code whose only non-standard dependency is that of a swapping operation on names. (An example of how this can be done for FreshML may be found in previous work[63, §3] by the author and others.) The provision for this in the language runtime system is thus essential. Some ML systems (O'Caml and Moscow ML, for example) permit the inspection and modification of the ML heap at runtime from within an ML program by using unsafe primitives and it has been shown that this can be used to implement a swapping 'primitive' without too much difficulty[2]. However, the solution is somewhat clumsy and prohibits the use of certain optimisations as we identify in *Efficiency* below.

► *Syntactic sugar.* If implementing freshness as a library it is highly desirable to have some pre-processing of language source text in order to provide idioms such as pattern-matching on abstraction values. These idioms in themselves are one of the distinguishing features of our approach: syntax-manipulating programs may be written concisely and elegantly using them. The source-to-source translations which one requires in order to expand out such constructs could possibly be implemented in a pre-processor such as `camlp4`[13]. However, past experience implementing (very) early versions of the FreshML system[59] showed that such transformations can be far from straightforward. This is particularly the case when dealing with the expansion of nested abstraction pattern-matches. Additionally, an upshot of such transformations can be the undue obfuscation of compiler error messages.

► *Efficiency.* In order to implement optimisations such the scheme of delayed permutations which will be introduced in §7.1.1, it is necessary to make fundamental changes to the structure of values on the heap. This obviously necessitates modification of the existing compiler and runtime system, rather than just adding a library module.

► *The moral stance.* Morally speaking, we feel that constructs for declaring and manipulating syntax modulo α-equivalence are so fundamental that they ought to be directly incorporated

[1] Miller[33, §1] presents a similar pair of options.

[2] Such a solution was first implemented by Claudio Russo in Moscow ML (private communication).

into the target language. Why should abstraction types be on any different a level from the other type constructions which we regularly use when declaring recursive datatypes?

▶ *The learning curve.* Wholesale extension of an existing language and compiler requires a significant amount of work, not just in understanding the semantics of the programming language but also in familiarisation with voluminous quantities of compiler code. In the case of the initial versions of Fresh O'Caml the lack of freshness inference meant that most of the modifications to the existing compiler code were relatively straightforward. The freshness-specific runtime code is fairly orthogonal to the rest of the runtime system and thus could be inserted into the system without undue difficulty. However as we shall see in this and the next chapter, the implementation is earmarked to become more refined with a view to improving efficiency. This will certainly involve more far-reaching changes affecting large volumes of compiler and runtime system code.

▶ *Accessibility.* The library solution has the great advantage that freshness features can be incorporated into some existing language on a relatively short timescale (even if only as a quick inefficient hack). It is therefore an attractive proposition for demonstrating the benefits of our approach to users of these languages. Such projects would hopefully increase general awareness and adoption of the new constructs.

6.2 System overview

The patch to Objective Caml is broadly composed of the following parts.

1. Modifications to the front-end of the compiler (lexer, parser and type-checker) to accommodate the new constructs. Whilst there are multitudinous changes, most are relatively minor. The major exception is the code which compiles abstraction pattern-matches, whose underlying algorithm is explained in §6.5.

2. A new module in the runtime system implementing low-level functionality (sources `byterun/freshness.c` and `byterun/freshness.h`).

3. Minor modifications throughout the runtime system and standard library to cope with the addition of new varieties of heap blocks (to represent abstraction values, for example) and primitive operations (for example atom-swapping).

The majority of the extensions are therefore in the runtime system which is implemented in C[26]; the remainder are written in O'Caml. The system remains a bootstrapping compiler and emits both bytecode and native code.

For readers unfamiliar with the internals of the O'Caml system, we must say a few words about how values are stored in memory during the execution of an O'Caml program. Values are partitioned into two varieties: those which are allocated on the heap (*boxed*) and those which are stored as immediate (*unboxed*) integer values.

Any particular allocated value will consist of one or more heap *blocks*. Each block has an integer tag which stores information about what is inside. For example, tag number zero specifies a tuple. Blocks may be linked together via pointers and may be aliased (shared) in order to save memory. They are garbage-collected when they are no longer referenced.

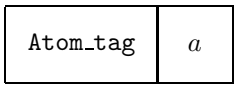
Immediate values are used to store values such as integers. On a machine whose native processor integer width is n bits, an O'Caml immediate value will have $n - 1$ bits available for actual data—the final bit being taken up with the flag which specifies whether or not the value is boxed.

6.3 Creation of fresh names

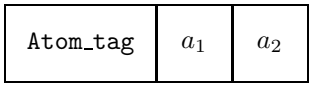
An object language name is represented by a block with tag `Atom_tag`. Such a block contains a single extra field containing an *atom identifier*, which is a 31-bit integer produced when the atom is created (corresponding to a `fresh` expression, for example). The integers are generated using a pseudo-random number generator[3] whose parameters are chosen to give a very low risk of collision. The reason for using such a linear congruential generator rather than just an incrementing counter is to attempt to ensure the uniqueness of identifiers across multiple

[3]Thanks to Keith Wansbrough for suggesting this.

instances of the O'Caml runtime, for there is currently no specific support for marshalling atoms. Future versions will likely use a simple counter and be equipped with such marshalling support. So currently, a heap block representing an object language name looks like[4]:



In order to minimise the risk of collisions (in the current implementation) or 'running out of atoms' (in future versions), it is highly desirable that atom identifiers should have a range[5] of 63 bits. In order to approximate this, on a 32-bit machine the heap layout could be changed to:



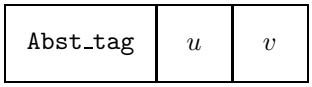
where a_1 (resp. a_2) is the most (resp. least) significant part of the atom identifier, or vice-versa. These fields may accommodate at most 31 bits each, so we obtain a 62-bit range. On a 64-bit machine, heap block fields are 64 bits wide by default so no change is needed. This will allow a 63-bit range for the atom identifiers.

6.4 Creation of abstraction values

As we noted in Chapter 2, creation of abstraction values is fast whereas deconstruction is slower. In the implementation, the creation of an abstraction value <<u>>v involves just two steps:

1. the allocation of a heap block with a distinguished tag (`Abst_tag`);
2. the placing of u and v inside as if for a pair.

Note that either of u and v may be immediate values, or pointers to boxed values. An abstraction heap block therefore looks like this:



6.5 Pattern-matching on abstraction values

It is well-known that the efficient compilation of pattern-matching[68] in functional languages is a non-trivial problem. In Fresh O'Caml we have extended the compiler to accommodate matches against abstraction values as described in §2.2.3. This poses a significant implementation hurdle—especially since the existing code to compile matches in O'Caml is highly optimised[27].

Early in the development of Fresh O'Caml, efficiency was completely put aside in order to get a working system as fast as possible. As a consequence, a temporary pattern-matching scheme was put in place which required only minimal modifications to the compiler code. We shall review this first and then describe more efficient implementation strategies.

Suppose we have a function which takes an arbitrary O'Caml value and determines whether there are any abstraction values present within it. Then, whenever compiling a match we emit code to intercept the value to be matched against and pass it through this function. If the value may possibly contain abstractions, then we completely freshen the value such that every abstraction value therein contains fresh atoms corresponding to those in binding position. The match may then be correctly compiled by recursively changing the abstraction patterns to pair patterns and passing the rewritten match to the normal pattern-match compilation code.

Now this scheme is obviously quite inefficient but it suffices for a first attempt. To improve upon this requires more substantial compiler modifications. When considering such modifications, we have aimed to minimise the amount of modifications to the central O'Caml pattern-matching code itself. To this end, current versions of Fresh O'Caml use what is effectively

[4] In case the reader is wondering, the tag field is indeed drawn disproportionately large.

[5] In the current implementation they are still restricted to 31 bits.

a source-to-source translation performed just before matches are compiled. It operates as follows.

Define a function $pat \mapsto [\![pat]\!]$ which rewrites a Fresh O'Caml pattern pat such that each abstraction pattern `<<`p_1`>>`p_2 is replaced by the corresponding pair pattern `(`p_1`,` p_2`)`. Similarly, define a function $pat \mapsto [\![pat]\!]_w$ which rewrites a pattern pat such that every pattern variable within it is replaced by a wildcard. Then we transform a (top-level) pattern match of the form

$$\texttt{<<}pat\texttt{>>}pat' \texttt{ -> } exp$$

into the following code fragment[6], for a fresh pattern variable x:

$$[\![\texttt{<<}pat\texttt{>>}pat']\!]_w \texttt{ as } x \texttt{ ->}$$
$$\texttt{ match } freshen\ x \texttt{ with } [\![\texttt{<<}pat\texttt{>>}pat']\!] \texttt{ -> } exp.$$

The function $freshen$ performs the 'complete freshening' operation on a value as described above. We must note that this operation is potentially very expensive—especially if a term is subjected to repeated matches down through its structure, where $O(n^2)$ time complexity may occur as a result of the swapping process. The scheme of *delayed permutations* which we shall examine in both the next section and in full detail in §7.1.1 will, however, alleviate this.

The obvious thing to note about our current scheme is that it is somewhat eager, in the sense that a single abstraction pattern-match causes the whole of the corresponding abstraction body to be freshened. A more incremental approach could be considered, whereby only the parts of the value being matched against using abstraction patterns get freshened. This saves the computational cost of the $freshen$ function. For example, for atoms a, b then we could consider matching the value `<<`a`>>(`a`, <<`b`>>`b`)` against the pattern `<<`p`>>(`q`,` r`)`. In the current scheme, the innermost abstraction (bound to r) would be fully freshened—even though to actually look inside it will require another abstraction pattern-match. In an incremental scheme, this innermost abstraction would not be freshened (or even copied) in the first instance.

An incremental scheme requires significantly more implementation effort, either via a complex source-to-source translation or via serious modification of the existing O'Caml pattern-matching compiler code. Future work should conduct some experiments on large-scale systems to determine whether the potential increases in efficiency offered by the incremental approach are worth the extra complexity inside the compiler.

6.6 Implementation of swapping

The crux of our implementation is the runtime swapping of atoms throughout arbitrary values. Even excepting use of the explicit atom-swapping construct `swap` e `and` e' `in` e'', there will likely be a very high number of transpositions required during the execution of the average syntax-manipulating program written in Fresh O'Caml. The majority (or even all) of these will arise from the deconstruction of abstraction values.

We can subdivide the general problem of efficient implementation of swapping into two separate, but related problems:

- how do we know when we *must* perform a swap; and
- if a swap is essential, how do we make that operation *fast*?

The following two sections consider these issues in turn.

6.6.1 When to swap?

It is more difficult than it might at first seem to give a good answer to the question of when to swap, especially when that answer must be reflected in compiler code! Let us just consider the case of abstraction pattern-matches, which is the main area of concern. Here are some partial code fragments which exhibit some possible situations, for a data constructor `C` and a function `F`. In fragments 1 through 3, no freshening needs to be performed on the value arriving to be matched. In fragments 4 and 5, freshening needs to be performed but at different times during evaluation.

[6] The astute reader will note that the transformation is slightly evil—the resulting code does indeed only work successfully because the heap layout of abstraction values coincides (modulo the tag) with that for pairs.

1. The right-hand side of an abstraction match does not contain occurrences of the pattern variables.

   ```
   let f1 t = match t with
     ... | C (<<a>>x) -> true | ...
   ```

   Such situations can be identified by a simple static analysis and the compiler can omit the emission of freshening code.

2. An abstraction pattern contains a wildcard.

   ```
   let f2 t = match t with
     ... | C (<<a>>_) -> ... | ...
   ```

   In this situation, we may still have to choose as many fresh atoms as there are atoms in the algebraic support of the value in binding position (here corresponding to the pattern variable `a`), even if we can avoid swapping. There are two possible cases:

   - A 'binding position only' pattern of the form `<<a>>_`. Here, the programmer does not have access to the body so we will never need to perform any transpositions on that part of the incoming value. However, since `a` could potentially be used by the programmer[7], it may be necessary to pick fresh atoms and perform some swaps in order to freshen the binding part of the incoming value.
   - A 'body position only' pattern of the form `<<_>>x`. In this case, the wildcard might as well not be there as the programmer may still make use of `x` in a situation where it needs freshening. Thus such patterns can be treated by picking a fresh pattern variable for the binding position and proceeding as normal.

3. The right-hand side of an abstraction match is such that no freshening of the incoming value needs to occur. For simplicity we consider `a` to be of bindable type.

   ```
   let f3 t = match t with
     ... | C (<<a>>x) -> (<<a>>x, <<a>>x) | ...
   ```

   Informally speaking, the reason why we do not need to freshen in this specific case is because the result of the computation on the right-hand side of the match (the pairing operation) is well-defined no matter which atom is bound to `a`. Formally speaking, we do not need to freshen because the atom bound to `a` will never occur in the support of the denotation of the value obtained by evaluating the right-hand side of the match. This means that we do not have to allocate any fresh atoms upon the match—thus reducing turnover of atoms.

   We approximate[8] such a judgement by a *freshness judgement* `a # (<<a>>x, <<a>>x)`, where the `#` is read as 'fresh for'. In the FreshML-2000 design[49], such judgements were an integral part of the static type system as it was thought that they were needed in order to ensure correctness properties such as those proved in Chapter 5. However, this turns out not to be the case.

   Section §6.8 examines freshness inference in detail. For now let us observe that it is a technology not currently exploited inside the Fresh O'Caml compiler but which could be potentially useful for this optimisation.

4. An abstraction match whose incoming value must always be freshened, but not necessarily straight away.

   ```
   let f4 t = match t with
     ... | C (<<a>>x) -> F x | ...
   ```

[7] This can be used to obtain the same effect as a `fresh` expression.

[8] We are running into problems of undecidability here, as we will discuss shortly.

Here, we may not be able to guarantee that `F` never uses its argument. However, the act of just *invoking* the function `F` does not require access to the innards of `x`. Therefore, it would suffice to mark the value bound to `x` as having a *pending* permutation of atoms attached to it. When (and if) `F` uses the value, the permutation must actually be applied through the value as far as necessary to ensure all used parts are freshened.

Situations like this are very common and theory has been developed—starting with an idea by Mark Shields (private communication)—to reason about such pending permutations. We shall examine this further in §7.1.1.

5. An abstraction match whose incoming value requires immediate freshening: there is no option.

```
let f5 t = match t with
  ... | C (<<a>>a') -> a = a' | ...
```

This code fragment could be rewritten using the O'Caml `when` qualification: another simple example of when the 'immediate freshening' situation could occur.

```
let f5 t = match t with
  ... | C (<<a>>a') when a = a' -> true | ...
```

Apart from abstraction pattern-matches, the only other construct whose evaluation involves swapping of atoms is `swap` e `and` e' `in` e''. In this case, the permutation may be able to be delayed depending on the surrounding context. Again, an optimisation can be made if the value to which e'' evaluates does not in fact contain any atoms.

6.6.2 How to swap?

Having discussed when a swapping operation needs to be performed, we now turn to the issue of actually performing such an operation. Here, we assume the absence of pending permutations: see §7.1.1 for evaluation strategies in that setting.

In general, we will have two lists of atoms $S = \{a_1, \ldots, a_n\}$, $S' = \{b_1, \ldots, b_n\}$ of equal length whose elements are to be exchanged simultaneously (in the sense that for each $1 \leqslant k \leqslant n$, a_k is to be exchanged with b_k) throughout some value v. At runtime, each atom will correspond to a heap block (viz. §6.3) containing an atom identifier. We thus traverse through the value v, copying the structure as we go[9], looking out for atom blocks whose identifiers which correspond to any in S or S'. Upon finding a match (say the atom a_k), we change the particular atom pointer in v to point at the new atom (in this case b_k). Note that we are only changing pointers to atoms, not performing in-place rewriting of atom identifiers.

A possible optimisation is to observe that in many cases, one of the lists (say S') may in fact solely consist of atoms which are fresh for the current environment. (This usually occurs as a result of abstraction pattern-matching where it has been determined that fresh atoms must be allocated.) In this case the swapping is actually only a 'fresh renaming'. Thus, upon encountering an atom in v we only need to look through the other list (in this case S) to determine if a swap must be performed upon a particular atom.

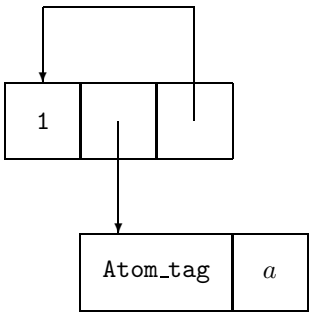
It may also be worth considering methods of simplifying lists of atom-swappings into smaller ones—although it is not clear whether this will give any significant improvement in efficiency. Future benchmarking will be required on this front.

So everything about *doing* swapping sounds fairly straightforward so far. Unfortunately, there is a thorn: (immutable) values on the O'Caml heap may be cyclic. Here is a fragment of an interactive session as a somewhat contrived, but illuminating example.

```
# let x = fresh;;
val x : 'a name = name_0
# let rec xs = x :: xs;;
val xs : 'a name list =
  [name_0; name_0; name_0; name_0; name_0; name_0; name_0;
   name_0; name_0; ...]
```

[9] Potentially highly inefficient: we say some words about this in §6.7.

The heap structure of the value `xs` would be as shown in the diagram below, where a is the atom identifier bound to the value identifier `x`. (The tag `1` corresponds to the list cons data constructor.)



In order to cope with atom-swapping throughout such values, we could attempt to use a hash table. Upon encountering a block as we traverse the heap, we check to see if its address k exists as a key in the hash table. If it does not, then it is a previously-unseen block; we perform the swapping operation upon it and record the address of the resulting value[10] v in the hash table, $k \mapsto v$. Conversely, if the address k was found in the hash table then we just use the corresponding value v in the hash table as the new value (without delving any deeper into the block k). Note that it is essential to use a data structure with $O(1)$ lookup, such as a hashtable, to prevent massive performance degredation.

Unfortunately, the actual C implementation of this algorithm is far from straightforward. This is because during the process of swapping we are very likely to allocate new values on the O'Caml heap. Each such allocation triggers a minor garbage collection, which may cause heap blocks to be moved. This is problematic since we are attempting to *key* a hashtable on the addresses of such blocks. Even if we can determine which blocks have been moved, then we need to re-shuffle the hash table as the particular invariant mapping keys to entry positions in the table will have been broken.

Instead of performing such operations (which are not only costly but also tricky to implement), we adopt an alternative multi-pass solution. In the first pass, we scan the value under question and determine precisely which heap allocations will need to be performed. This is the only stage which actually requires the hash table in order to prevent looping. After the first stage, we have an array which not only gives the sizes of the allocations required but also indicates which blocks should be shared. In the second stage, we perform the allocations. Then in the third stage we traverse the value in exactly the same order as the first stage, at the same time iterating through the array to determine what to do with the next block which we encounter. At this point, we can perform the actual permutation—or if the block is shared with one encountered previously then we just use the previously-computed permutation of that block.

This procedure ensures that the swapping algorithm does not loop on cyclic values. Here is a short continuation of the previous session.

```
# let y = fresh;;
val y : 'a name = name_0
# match (swap x and y in xs) with b::bs -> (b = x, b = y);;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
- : bool * bool = (false, true)
```

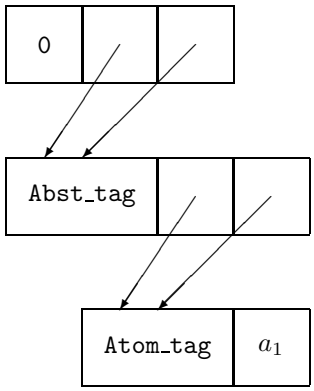
6.7 Preservation of sharing

An important concept in the implementation of functional languages is that of *sharing*: the re-use, possibly multiple times, of existing heap blocks to prevent unnecessary duplication. For example, the expression

[10] One would hope that this address would be equal to that of k if there are in fact no atoms occurring inside the block k.

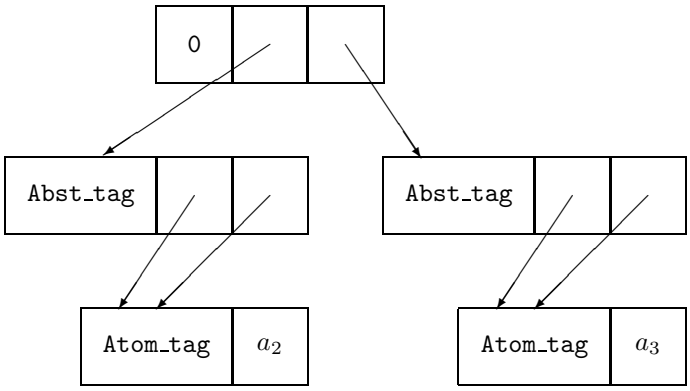
```
let a = fresh in let x = <<a>>a in (x, x)
```

will likely yield the following heap structure:



where a_1 is the atom identifier bound to the value identifier `a`. Note how we only use three instead of seven heap blocks as one might at first imagine.

When manipulating the heap, it is desirable to preserve as much of this sharing as possible. Therefore when traversing a value in order to perform transpositions we should not copy unnecessarily. Unfortunately, when performing such a traversal we may *have* to un-share blocks (via copying) in order to produce a correct result. For example, if the above value is to be freshened completely then we must make sure that *two* atoms are allocated (assuming that the existing identifier a_1 is not sufficiently fresh—if it were to be, we must still allocate another). Then, each component of the pair value must point to a *different* abstraction block like so:



where a_2 and a_3 are fresh atom identifiers. Note how despite the freshening some sharing has still been preserved. However at the time of writing the Fresh O'Caml system does not implement such a scheme: it is a matter for later implementation work. With delayed permutations of course, necessary un-sharing will only happen at the last minute—thus helping further with memory efficiency.

6.8 Freshness inference

6.8.1 The motivation for a static analysis

In §6.6.1 we saw an example of a *freshness judgement*; we postulated that this could be exploited in order to perform an optimisation. Such freshness judgements can be produced using a certain static analysis known as *freshness inference* which is performed in parallel with type-checking. In this section we shall examine this analysis in detail, mostly recalling earlier work[49] but adding some extra discussion and insights gained from implementing a system[11] which made use of the analysis. When designing that system, it was thought that freshness

[11] The FreshML-2000 interpretive system.

inference was necessary in order to obtain correctness properties such as those in Chapter 5. It was only the work of Cardelli et al.[9] which caused the first doubts to be raised about it—leading to its subsequent deprecation.

As we noted in Chapter 1, we often use the Barendregt variable convention to reason about programs operating on α-equivalence classes of object language syntax: we simply choose 'suitably fresh' names at runtime when necessary. Informally speaking, the result of such a computation appears to be independent of *which particular* bound names have been chosen just because the denotation of the final value does not include those names in its support. When computing in Fresh O'Caml, these 'bound names' are atoms in binding position and so we should be interested in whether results of computations include certain atoms in the denotation of their supports. If the denotation of a result contains an atom in its support which has been dynamically generated at runtime (say by evaluation of `fresh`), then that effect will be visible to any surrounding context. If the atom is not in the support of the denotation of the result, then no side-effects arising from the generation of the atom will be visible.

The aim of freshness inference is to build up a sound approximation to this notion of 'not in the support of' by performing a static analysis which yields a relation between value identifiers and expressions. A *freshness judgement* such as $x \# e$ (read 'x fresh for e') is intended to mean that e will always evaluate to a value v such that $a \notin \mathrm{supp}(\mathcal{V}[\![v]\!])$, where a is the atom bound to the value identifier x at runtime and $\mathcal{V}[\![v]\!]$ is the denotation of the value v.

The upshot of this analysis is that those program phrases which have observable effects of creating fresh atoms may be rejected at compile-time, thus making the dynamic allocation of atoms referentially transparent. (A convenient upshot of this is that reasoning about programs becomes more straightforward due to the lack of these side-effects.) For example, take once again the untyped λ-calculus with the corresponding Mini-FreshML datatype declaration from §5.7 and consider the expressions

```
let x = fresh in Lam(<<x>>x)  and  let x = fresh in Var(x).
```

The result of evaluating the first expression is a value `<<`a`>>`a (for some $a \in \mathbb{A}$) which clearly corresponds to the closed object language term $\lambda x.\ x$. The dynamic allocation of the atom a has no visible external effect, as no matter which a is chosen then the same value is obtained. However, the second expression simply evaluates to the representation of 'a fresh object language variable': *which particular* atom has been allocated may be determined from the outside. Writing $\neg\#$ to mean 'not fresh for', the procedure of freshness inference would yield the judgements that

```
x # let x = fresh in Lam(<<x>>x)
x ¬# let x = fresh in Var(x)
```

which would cause the type-checker to reject the second expression.

Another good example of the lack of referential transparency in Fresh O'Caml is the function `bound_vars` of §2.2.3, whose FreshML-2000 equivalent would have been rejected at compile time.

6.8.2 Static semantics with freshness inference

How is freshness inference actually performed? From a 'logical' rather than an 'inference' point of view, we can express the algorithm as a set of axioms and rules as illustrated in Figure 6.1. In that Figure, we see that the normal system of typing judgements for a language similar to Mini-FreshML has been extended with freshness judgements. In the new system, the standard typing relation $\vdash$ on 3-tuples (Γ, e, τ) is transformed to another on 5-tuples $(\Gamma, \nabla, e, \tau, X)$ where ∇ is a *freshness context* and X is a finite set of value identifiers. Freshness contexts are finite binary relations on value identifiers, typical element $x \# x'$. The intention here is to carry along with us some information about the freshness of one value identifier relative to another in order that we can use it at the top of the rule trees (at axiom (vid) in particular). The '_' symbol corresponds to a 'don't care'.

Before identifying the other notation used in the rules, note that the first component of each member of a freshness context (here x) must be of type `name`. For otherwise it makes no sense to consider whether the atom assigned to it at runtime may occur in the support of the denotation of the value assigned to x'. Furthermore, if we have $x \# x'$ contained within some

$$\text{vid}\,\frac{}{\Gamma;\nabla \vdash x \,:\, \tau \;\#\; \nabla(x)}\; x \in \mathrm{dom}(\Gamma) \text{ and } \tau = \Gamma(x)$$

$$\text{con}\,\frac{\Gamma;\nabla \vdash e \,:\, \sigma_k \;\#\; X}{\Gamma;\nabla \vdash \mathtt{C}_k\mathtt{(}e\mathtt{)} \,:\, \delta \;\#\; X} \qquad \text{pair}\,\frac{\Gamma;\nabla \vdash e \,:\, \tau \;\#\; X \quad \Gamma;\nabla \vdash e' \,:\, \tau' \;\#\; X}{\Gamma;\nabla \vdash \mathtt{(}e\mathtt{,}\ e'\mathtt{)} \,:\, \tau \times \tau' \;\#\; X}$$

$$\text{abst}\,\frac{\Gamma;\nabla \vdash x \,:\, \mathtt{name} \;\#\; \_ \quad \Gamma;\nabla \vdash e \,:\, \tau \;\#\; X}{\Gamma;\nabla \vdash \mathtt{<<}x\mathtt{>>}e \,:\, \mathtt{<<name>>}\tau \;\#\; X \cup \{x\}}$$

$$\text{swap}\,\frac{\Gamma;\nabla \vdash e_1 \,:\, \mathtt{name} \;\#\; X \quad \Gamma;\nabla \vdash e_2 \,:\, \mathtt{name} \;\#\; X \quad \Gamma;\nabla \vdash e_3 \,:\, \tau \;\#\; X}{\Gamma;\nabla \vdash \mathtt{swap}\ e_1\mathtt{,}\ e_2\ \mathtt{in}\ e_3 \,:\, \tau \;\#\; X}$$

$$\text{fun}\,\frac{\begin{array}{c}\Gamma, f \,:\, \tau \to \tau', x \,:\, \tau;\nabla \vdash e \,:\, \tau' \;\#\; \_ \\ \text{for all free variables } v \text{ of } e,\ v \neq f \ \wedge\ v \neq x \Rightarrow \Gamma;\nabla \vdash v \,:\, \_ \;\#\; X\end{array}}{\Gamma;\nabla \vdash \mathtt{fun}\ f\mathtt{(}x\mathtt{)}\ \mathtt{=}\ e \,:\, \tau \to \tau' \;\#\; X}$$

$$\text{app}\,\frac{\Gamma;\nabla \vdash e \,:\, \tau' \to \tau \;\#\; X \quad \Gamma;\nabla \vdash e' \,:\, \tau' \;\#\; X}{\Gamma;\nabla \vdash e\ e' \,:\, \tau \;\#\; X}$$

$$\text{let}\,\frac{\Gamma;\nabla \vdash e \,:\, \tau' \;\#\; X' \quad \Gamma, x \,:\, \tau';\nabla, x \;\#_\Gamma\; X' \vdash e' \,:\, \tau \;\#\; X}{\Gamma;\nabla \vdash \mathtt{let}\ x\ \mathtt{=}\ e\ \mathtt{in}\ e' \,:\, \tau \;\#\; X \setminus \{x\}}$$

$$\text{let-fresh}\,\frac{\Gamma, x \,:\, \mathtt{name};\nabla, x \;\#\; \Gamma \vdash e \,:\, \tau \;\#\; X}{\Gamma;\nabla \vdash \mathtt{let}\ x\ \mathtt{=}\ \mathtt{fresh}\ \mathtt{in}\ e \,:\, \tau \;\#\; X \setminus \{x\}}$$

$$\text{let-pair}\,\frac{\begin{array}{c}\Gamma;\nabla \vdash e \,:\, \tau_1 \times \tau_2 \;\#\; X' \\ \Gamma, x \,:\, \tau_1, x' \,:\, \tau_2;\nabla, x \;\#_\Gamma\; X', x' \;\#_\Gamma\; X' \vdash e' \,:\, \tau \;\#\; X\end{array}}{\Gamma;\nabla \vdash \mathtt{let}\ \mathtt{(}x\mathtt{,}\ x'\mathtt{)}\ \mathtt{=}\ e\ \mathtt{in}\ e' \,:\, \tau \;\#\; X \setminus \{x, x'\}}$$

$$\text{let-abst}\,\frac{\begin{array}{c}\Gamma;\nabla \vdash e \,:\, \mathtt{<<name>>}\tau' \;\#\; X' \\ \Gamma, x \,:\, \mathtt{name}, x' \,:\, \tau';\nabla, x \;\#\; \Gamma, x' \;\#_\Gamma\; X' \vdash e' \,:\, \tau \;\#\; X\end{array}}{\Gamma;\nabla \vdash \mathtt{let}\ \mathtt{<<}x\mathtt{>>}x'\ \mathtt{=}\ e\ \mathtt{in}\ e' \,:\, \tau \;\#\; X \setminus \{x, x'\}}$$

$$\text{match}\,\frac{\begin{array}{c}\Gamma;\nabla \vdash e \,:\, \delta \;\#\; X' \\ \text{for all } 1 \leq k \leq K,\ \Gamma, x_k \,:\, \sigma_k;\nabla, x_k \;\#_\Gamma\; X' \vdash e_k \,:\, \tau \;\#\; X\end{array}}{\Gamma;\nabla \vdash \mathtt{match}\ e\ \mathtt{with}\ \cdots\mathtt{|}\ \mathtt{C}_k\mathtt{(}x_k\mathtt{)}\ \mathtt{->}\ e_k\ \mathtt{|}\ \cdots \,:\, \tau \;\#\; X \setminus \{x_1, \ldots, x_K\}}$$

Figure 6.1: Static semantics with freshness inference.

freshness context ∇ then we may well be interested in whether $x' \;\#\; x$. This of course only makes sense if x' is of type `name`—but this is perfectly possible. Therefore, we must ensure that the freshness contexts are always 'symmetrised' after they are extended in order to expediate lookups. To do this we employ the following notation, which is used in the Figure.

- Given a finite set of value identifiers X we write $\nabla, x \;\#_\Gamma\; X$ (where the identifier x of type `name` occurs neither in X nor ∇) to indicate that freshness context judging $x \;\#\; y$ for each $y \in X$, $y \;\#\; x$ for each $y \in X$ such that $\Gamma(y) = \mathtt{name}$ and acting as ∇ otherwise.
- For clarity we write $\nabla, x \;\#\; \Gamma$ to mean $\nabla, x \;\#_\Gamma\; \mathrm{dom}(\Gamma)$.
- In rule (vid), we use the notation $\nabla(x)$ to indicate the finite set of value identifiers $\{x' \mid (x, x') \in \nabla\}$.

Note how we have to incorporate `let` x `=` `fresh in` e as a single binding construct rather than just providing a `fresh` expression as in Mini-FreshML. The reason for this is because it is necessary to obtain the exact value identifier which has been declared fresh in order that freshness judgements relating to that identifier may be produced. Similarly, we must insist that abstraction expressions must be of the form `<<`x`>>`e rather than `<<`e`>>`e'.

Having such constraints imposed on the syntax of the language is rather a nuisance. However in the old FreshML systems this was not so much of a problem, for the 'declara-

tion/expression' split present in the Standard ML language[36] (and not reflected in Mini-FreshML) fits quite well with freshness inference. In this setting, the `fresh` construct becomes a declaration rather than an expression. Such declarations then elaborate to an environment paired with a set of identifiers which should be treated as fresh throughout the scope of the declaration.

Let us now run through a few of the rules to see the logic behind them. For value identifiers, we simply use the current freshness context to determine which other identifiers are fresh for the one in question. Constructed values are similarly straightforward. However, pair values are more interesting and indeed better understood from an 'inference' point of view. Thinking in this way, we can re-jig the rules somewhat to better express how a freshness inference algorithm would work—just as we could when describing a traditional type system. For example, we could rewrite the rule for pairs as follows:

$$\text{pair}\frac{\Gamma;\nabla \vdash e \,:\, \tau \;\#\; X_1 \qquad \Gamma;\nabla \vdash e' \,:\, \tau' \;\#\; X_2}{\Gamma;\nabla \vdash (e,\ e') \,:\, \tau \times \tau' \;\#\; X_1 \cap X_2}$$

This rule tells us that in order to deduce what is fresh for a pair $(e,\ e')$, we calculate what is fresh for each component and then take the *intersection* of the two. This gives the maximal set of value identifiers fresh for all sub-expressions (this can be thought of as being analogous to the concept of 'most general unifier').

Such an algorithm inherently provides relatively coarse-grained freshness information, which could possibly be alleviated by using 'restricted types' $\tau \;\#\; X$ throughout the typing process. Recent work by Schöpp and Stark[56] presents an intriguing type system which makes use of such types with inbuilt freshness information, which they call 'freefrom types'. This (dependently-typed) system enables them to express abstract syntax both in 'FM' style as used in this thesis and in the style of higher-order abstract syntax.

When dealing with an abstraction expression, we simply type-check the body and then deem the identifier in binding position (call it x) to be fresh for the whole expression e. This corresponds to the evaluation strategy: at runtime, this identifier will correspond to a fresh atom a which occurs in binding position inside an abstraction value (call it <<a>>v) and therefore never occurs in the support of its denotation.

Finally, let us examine the rule for recursive functions. As one might expect, the decision of 'not in the support of' is undecidable at function types since it relies on extensional equality of functions (recall that a function f does not contain an atom a in its support just when for each $x \in \mathrm{dom}(f)$, $(a\ a') \cdot (f(x)) = f((a\ a') \cdot x)$ for some/any fresh atom a'). Therefore when performing freshness inference we need to produce a sound approximation in order to get a decidable static analysis. Unfortunately, as noted by Pitts and Gabbay[49], freshness is not a logical relation: just because an atom is fresh for a particular value v and also fresh for the result $f(v)$ does not mean that it is fresh for the function. This is due to the fact that functions can contain concealed atoms within their support. For example consider the function `fun` $f(x)$ `=` <<a>>x, where a is some value identifier mapping to an atom a. Whilst a is always fresh for the result of the function, it is not fresh for the function itself.

So in the procedure of freshness inference, we simply judge that if an identifier is fresh for all of the function's free identifiers then it is fresh for the function itself. This is a sound, albeit over-conservative, approximation which we have to live with.

The difficulty with this particular approximation is that it renders many apparently well-typed programs untypeable. (Indeed, many programs that *really do* satisfy the necessary conditions are not passed by the inference algorithm.) A good example is provided by the following function which we saw in Chapter 2. It calculates the free variables of a PLC term.

```
let rec free_vars t =
  match t with
    Tvar x -> [x]
  | Tlam (ty, <<x>>t') -> remove x (free_vars t')
  | Tgen (<<a>>t') -> free_vars t'
  | Tapp (t1, t2) -> (free_vars t1) @ (free_vars t2)
  | Tspec (t', ty) -> free_vars t';;
```

Due to the semantics of `remove`, any atom assigned to the pattern variable `x` inside the `Tlam` clause will always be fresh for the right-hand side of the match. Unfortunately, the

freshness inference algorithm cannot deduce this—-which would mean that the FreshML-2000 version of this function would be rejected. Intuitively, we can see that the function will be thrown out because it requires *evaluation* to determine that `x` is never in the support of `remove x (free_vars t')`. To get around this problem, one can conceal the atom generated by the pattern-match on `Tlam` inside an abstraction, as follows.

```
let rec free_vars t =
  match t with
    Tvar x -> [x]
  | Tlam (ty, <<x>>t') -> remove (<<x>>(free_vars t'))
  | Tgen (<<a>>t') -> free_vars t'
  | Tapp (t1, t2) -> (free_vars t1) @ (free_vars t2)
  | Tspec (t', ty) -> free_vars t';;
```

This of course necessitates the refactoring of the `remove` function to have type `<<'a>>('a list) -> 'a list`. How grotesque.

6.8.3 Purity analysis

In order to produce anything near a sensible approximation to the notion of 'not in the support of', the freshness inference algorithm must proceed not only on the structure of expressions but also on the structure of *inferred types* as well[12]. As an example consider the following function, defined for some arbitrary expression e of type `unit` $\times$ `unit`.

$$\texttt{let f p = match p with <<a>>x ->}\ e.$$

Due to the recursion-theoretic problems which we described in the previous section, there is no way that a plain freshness inference algorithm can deduce that the atom bound to `a` is always fresh for e. The idea of purity checking is to make deductions of the form

> *no atoms can ever occur in a value of type* `unit` $\times$ `unit`*, so I know that* $x \ \#\ e$ *for each identifier* x *in scope.*

We say that types such as `unit` $\times$ `unit` are *pure*. With such judgements, we can extend the freshness inference algorithm to accept more expressions. In order to make this of much use, however, the procedure of purity checking must also cope when the type in question is a user-defined, possibly-recursive datatype. How can one deduce whether atoms ever occur inside a value of such a type? We invented a solution analogous to that used by Standard ML[36] to compute whether such a type admits equality. We start by asserting that values of each of the datatypes being (mutually) declared may never contain atoms. Then, one-by-one until there are no more changes, we remove any such judgement which may be contradicted by examining the types of the various data constructors. In this way, we converge on a greatest fixed point.

Since FreshML permits multiple user-defined types of bindable names, we also refine the judgement a little further so we can deduce that values of some datatype δ may never contain occurrences of atoms of bindable type τ, but might possibly contain occurrences of atoms of some other bindable type τ'. Improvements such as these increase the number of programs accepted by the compiler and were found to be essential in practice. However, when a program cannot be accepted it can be difficult for the programmer to understand just why this is the case: the combination of type inference, freshness inference and purity analysis can be formidable.

6.8.4 A note on denotational semantics

We should observe that freshness judgements inside a context ∇ may be reflected in a denotational setting by taking subsets of the usual denotations of typing contexts. For we can define the FM-cppo

$$[\![\Gamma; \nabla]\!] \stackrel{\text{def}}{=} \{\rho \in [\![\Gamma]\!]_{\downarrow} \mid \forall (x, x') \in \nabla.\ \rho(x) \notin \mathrm{supp}(\rho(x'))\} \cup \{\bot\}$$

[12] This causes problems when polymorphic types are present in the language, as they can inhibit the inference of any purity information at all.

which is a subset of $[\![\Gamma]\!]$ (and inherits its permutation action). As an item of possible future work, we postulate that freshness judgements could be incorporated into our continuation-based semantics by providing functions

$$\begin{array}{rcl} \mathcal{V}[\![\Gamma;\nabla \vdash v : \tau \# X]\!] & \in & [\![\Gamma;\nabla]\!] \multimap [\![\tau]\!] \\ \mathcal{S}[\![\Gamma;\nabla \vdash_s S : \tau \multimap \_ \# X]\!] & \in & [\![\Gamma;\nabla]\!] \multimap [\![\tau]\!]^{\perp} \\ \mathcal{E}[\![\Gamma;\nabla \vdash e : \tau \# X]\!] & \in & [\![\Gamma;\nabla]\!] \multimap [\![\tau]\!]^{\perp\perp}. \end{array}$$

However, it is not clear that such an extended semantics would be particularly useful for proving correctness properties (unless freshness inference makes it back into the language, which is very unlikely). A more likely application would be to validate the correctness of any optimisations which might make use of freshness inference.

6.8.5 An abstraction monad

An interesting stepping-stone along the way to discarding freshness inference was the discovery that a monadic style of programming could be used to conceal the side-effects of generating fresh names. The original motivation for wanting such a monad was simply to get FreshML-2000 programs through the type-checker.

The monad may be encoded in Fresh Objective Caml[13] in the style of a Kleisli triple, thus:

```
type ('a,'b) abst = AMunit of 'b | AMabst of <<'a name>>(('a,'b) abst);;

let am_unit v = AMunit v;;
let am_let d f =
  match d with
    AMunit v -> f v
  | AMabst (<<a>>d') -> AMabst (<<a>>(am_let d' f));;
```

An alternative (and, we suggest, more readable) encoding could make use of a more general abstraction type as follows.

```
type ('a,'b) abst = AMabst of <<'a name list>>'b;;
let am_let =
  function (AMabst (<<xs>>v)) ->
  function f ->
    match (f v) with AMabst (<<ys>>v') -> AMabst (<<xs @ ys>>v');;
```

The `am_let` function has the following type as would be expected.

```
am_let : 'a, 'b abst -> ('b -> ('a, 'c abst)) -> ('a, 'c abst)
```

We can now conceal name generation by using the monad's data constructors to wrap up dynamically allocated names, coupled with `am_let` to sequence side-effecting computations. However, if multiple types of bindable names are involved we have to extend the monad in order to cope. It is also convenient to add some extra helper functions corresponding to the monad unit and multiplication. For example:

```
type ('a,'b,'c) abst = AMabst of <<'a name list * 'b name list>>'c;;

let am_unit v = AMabst (<<[],[]>>v);;
let am_mult (AMabst (<<xs,ys>>(AMabst (<<xs',ys'>>v)))) =
  AMabst (<<xs@xs',ys@ys'>>v);;
let am_let =
  function (AMabst (<<xs,ys>>v)) ->
  function f ->
    match (f v) with AMabst (<<xs',ys'>>v') ->
      AMabst (<<xs@xs',ys@ys'>>v');;
```

[13] Originally, we were working with Standard ML-style syntax—but we stick to Fresh O'Caml syntax here for consistency with the rest of this thesis.

Using this extended monad, we can now present a new version of `free_vars` (the original of which is presented on page 97) which would get through freshness inference.

```
let rec free_vars t =
  match t with
    Tvar x -> am_unit [x]
  | Tlam (ty, <<x>>t') ->
      am_mult (
        AMabst(<<[x],[]>>
        (am_let (free_vars t')
                (fun v ->
                 AMabst (<<[x],[]>>(remove x v))))))
  | Tgen (<<a>>t') ->
      am_mult (AMabst (<<[],[a]>>(free_vars t')))
  | Tapp (t1, t2) ->
      am_let (free_vars t1) (fun v1 ->
      am_let (free_vars t2) (fun v2 ->
        am_unit (v1 @ v2)))
  | Tspec (t', ty) -> free_vars t';;
```

The type of `free_vars` is now as follows. Note the effect of using generative type declarations such as those in §2.1.1: the arguments of the type constructor `name` may be easily confused.

```
free_vars : term -> (t, t, var list) abst
```

We leave it to the reader to decide for themselves which is the most attractive gargoyle: this fragment or the refactored one on page 98.

7 Conclusions and future work

'We can only see a short distance ahead, but we can see plenty there that needs to be done.' —Turing

It is now time to take a step back and assess the work in this thesis. We believe that our major contributions have been:

- the development of a full-scale language, Fresh O'Caml, for metaprogramming with bindable names;
- the development of FM-domain theory, for reasoning about names and name binding;
- a denotational semantics using frame-stack semantics and a monad of continuations to model name generation;
- a detailed proof of correctness properties for Mini-FreshML.

So far, Fresh O'Caml has proved itself very useful for the rapid prototyping or first implementation of syntax-manipulating algorithms. However, much remains to be done. The implementation is not yet in a production-ready state and there are still some significant issues to be resolved. These include the improvement of efficiency, which we shall address again later in this chapter, and language enhancements such as a more pragmatic treatment of reference cells (viz. §2.6).

We believe that the mathematical theory developed in Chapter 4 is more generally-applicable beyond the scope of a denotational semantics for a Mini-FreshML-like language. There are already two examples of wider application within the literature: that which we cited in §1.2 by Abramsky et al.[2] which applies FM-domain theory to the world of game semantics; and that by Benton and Leperchey[6] which applies it to reason about mutable store.

The denotational semantics which we presented in Chapter 5 is, however, is rather more specialised to our particular setting. Whilst we do claim that it is good for proving properties such as those we have seen, there are clearly open questions—particularly in the areas of full abstraction and the validation of certain troublesome equivalences such as those mentioned in §5.6. However, use of the continuation monad to model name generation is most definitely a generally-applicable tool to use in conjunction with FM-domain theory—see [6] again for example.

7.1 Future work

7.1.1 Delayed permutations

We noted in §6.6.1 that a scheme of *delayed permutations* (somewhat reminiscent of [1]) could be used to improve efficiency—indeed, this is one of the major speed improvements pending implementation in Fresh O'Caml. In this section we examine how delayed permutations could be introduced to Mini-FreshML, working from an idea originally due to Mark Shields (private communication). The presentation is much simplified by adopting an environment-style semantics (described in §3.5).

What we are going to do is to partition the canonical forms into two varieties: those equipped with a *pending permutation* which is manifest on the outside and those without. Canonical forms in the second category may still contain pending permutations deeper within

their structure; but they may be immediately examined at the top level without any atom-swapping being required. So the new grammar of values is as follows:

$$\begin{array}{rcll} v & ::= & \texttt{()} & \text{unit} \\ & | & a & \text{atom} \\ & | & \mathtt{C}_k\texttt{(}v_p\texttt{)} & \text{data construction} \\ & | & \texttt{(}v_p\texttt{, }v_p\texttt{)} & \text{pairing} \\ & | & \texttt{<<}a\texttt{>>}v_p & \text{abstraction} \\ & | & [E, \texttt{fun } x\texttt{(}x\texttt{) = }e] & \text{recursive function closure} \\ v_p & ::= & \pi * v & \text{value with delayed permutation} \end{array}$$

where π stands for a permutation generated like so:

$$\begin{array}{rcll} \pi & ::= & [] & \text{null permutation} \\ & | & (a\ a') :: \pi & \text{atom-swap} \end{array}$$

and where E is a finite map from value identifiers to *values with delayed permutations* v_p. We write $\pi(a)$ to indicate the bona-fide application of a permutation to an atom. Given a value with delayed permutation $v_p = \pi * v$ and any permutation π' we write $\pi' * v_p$ to be the value $(\pi' \mathbin{@} \pi) * v$, where $\pi' \mathbin{@} \pi$ is that permutation whose action first permutes as for π and then as for π'.

The grammar for (non-canonical) expressions is as in Figure 3.1, with the exception that naked atoms a no longer appear. (This is a pleasing consequence of using an environment-style semantics: the canonical forms do not have to be a subset of the expressions. If using substituted-in style, then the necessity to delay permutations throughout closures means that the grammar for non-canonical forms needs extending with delayed permutations as well: this rapidly becomes very difficult to deal with.)

The semantics makes use of an auxiliary function $\mathrm{cf}(\text{–})$ which takes a value with delayed permutation v_p and pushes the permutation through the first level of its structure, thus making the outermost constructor manifest. This transformation is defined as follows:

$$\begin{array}{rcl} \mathrm{cf}(\pi * \texttt{()}) & \stackrel{\text{def}}{=} & \texttt{()} \\ \mathrm{cf}(\pi * a) & \stackrel{\text{def}}{=} & \pi(a) \\ \mathrm{cf}(\pi * \mathtt{C}_k\texttt{(}v_p\texttt{)}) & \stackrel{\text{def}}{=} & \mathtt{C}_k\texttt{(}\pi * v_p\texttt{)} \\ \mathrm{cf}(\pi * \texttt{(}v_p\texttt{, }v'_p\texttt{)}) & \stackrel{\text{def}}{=} & \texttt{(}\pi * v_p\texttt{, }\pi * v'_p\texttt{)} \\ \mathrm{cf}(\pi * \texttt{<<}a\texttt{>>}v_p) & \stackrel{\text{def}}{=} & \texttt{<<}\pi(a)\texttt{>>}\pi * v_p \\ \mathrm{cf}(\pi * [E, \texttt{fun } f\texttt{(}x\texttt{) = }e]) & \stackrel{\text{def}}{=} & [\{\,(x, \pi * v_p) \mid (x, v_p) \in E\,\}\texttt{, fun } f\texttt{(}x\texttt{) = }e]. \end{array}$$

Figure 7.1 provides a revised big-step semantics using an evaluation relation $\Downarrow_{\mathrm{p}}$ on 5-tuples $(\overline{a}, E, e, v, \overline{a}')$, where $\overline{a}$, $\overline{a}'$ are finite sets of atoms, E is an environment as described above, e is an expression and v a value *with its outermost constructor manifest*. We write $\overline{a}, E \vdash e \Downarrow_{\mathrm{p}} v, \overline{a}'$ iff $(\overline{a}, E, e, v, \overline{a}')$ is in the relation.

Note that the procedure of purity analysis discussed in §6.8.3 could be applied to discard unnecessary pending permutations. For given a value $\pi * v$ of type τ then we can simplify it to $[] * v$ if τ is pure. However, implementation of this will require more type information in the runtime heap than is present when using current versions of Fresh O'Caml.

We now wish to relate the new scheme of evaluation to the original one used in Chapter 5. To do this, define the function erase mapping values with their outermost constructor manifest to canonical forms containing no permutations (in the sense of Figure 3.1) as follows.

$$\begin{array}{rcl} \mathrm{erase}(\texttt{()}) & \stackrel{\text{def}}{=} & \texttt{()} \\ \mathrm{erase}(a) & \stackrel{\text{def}}{=} & a \\ \mathrm{erase}(\mathtt{C}_k\texttt{(}v_p\texttt{)}) & \stackrel{\text{def}}{=} & \mathtt{C}_k\texttt{(}\mathrm{erase}(\mathrm{cf}(v_p))\texttt{)} \\ \mathrm{erase}(\texttt{(}v_p\texttt{, }v'_p\texttt{)}) & \stackrel{\text{def}}{=} & \texttt{(}\mathrm{erase}(\mathrm{cf}(v_p))\texttt{, }\mathrm{erase}(\mathrm{cf}(v'_p))\texttt{)} \\ \mathrm{erase}(\texttt{<<}\pi * a\texttt{>>}v_p) & \stackrel{\text{def}}{=} & \texttt{<<}\pi(a)\texttt{>>}\mathrm{erase}(\mathrm{cf}(v_p)) \\ \mathrm{erase}([E, \texttt{fun } f\texttt{(}x\texttt{) = }e]) & \stackrel{\text{def}}{=} & [\{\,(x, \mathrm{erase}(\mathrm{cf}(v_p))) \mid (x, v_p) \in E\,\}\texttt{, fun } f\texttt{(}x\texttt{) = }e]. \end{array}$$

We now claim the following, whose proof is left to future work.

$$\text{vid}\frac{}{\overline{a}, E \vdash x \Downarrow_{\mathrm{p}} \mathrm{cf}(E(x)), \overline{a}}\ x \in \mathrm{dom}(E) \qquad \text{unit}\frac{}{\overline{a}, E \vdash \texttt{()} \Downarrow_{\mathrm{p}} \texttt{()}, \overline{a}}$$

$$\text{con}\frac{\overline{a}, E \vdash e \Downarrow_{\mathrm{p}} v, \overline{a}'}{\overline{a}, E \vdash \mathtt{C}_k\texttt{(}e\texttt{)} \Downarrow_{\mathrm{p}} \mathtt{C}_k\texttt{(}[] * v\texttt{)}, \overline{a}'} \qquad \text{fresh}\frac{}{\overline{a}, E \vdash \texttt{fresh} \Downarrow_{\mathrm{p}} a, \overline{a} \uplus \{a\}}\ a \in \mathbb{A} \setminus \overline{a}$$

$$\text{pair}\frac{\overline{a}, E \vdash e \Downarrow_{\mathrm{p}} v, \overline{a}' \qquad \overline{a}', E \vdash e' \Downarrow_{\mathrm{p}} v', \overline{a}''}{\overline{a}, E \vdash \texttt{(}e\texttt{, }e'\texttt{)} \Downarrow_{\mathrm{p}} \texttt{(}[] * v\texttt{, }[] * v'\texttt{)}, \overline{a}''}$$

$$\text{abst}\frac{\overline{a}, E \vdash e \Downarrow_{\mathrm{p}} a, \overline{a}' \qquad \overline{a}', E \vdash e' \Downarrow_{\mathrm{p}} v, \overline{a}''}{\overline{a}, E \vdash \texttt{<<}e\texttt{>>}e' \Downarrow_{\mathrm{p}} \texttt{<<}[] * a\texttt{>>}[] * v, \overline{a}''}$$

$$\text{swap}\frac{\overline{a}, E \vdash e \Downarrow_{\mathrm{p}} a, \overline{a}' \qquad \overline{a}', E \vdash e' \Downarrow_{\mathrm{p}} a', \overline{a}'' \qquad \overline{a}'', E \vdash e'' \Downarrow_{\mathrm{p}} v, \overline{a}'''}{\overline{a}, E \vdash \texttt{swap }e\texttt{, }e'\texttt{ in }e'' \Downarrow_{\mathrm{p}} \mathrm{cf}((a\ a') :: [] * v), \overline{a}'''}\ a, a' \in \mathbb{A}$$

$$\text{fun}\frac{}{\overline{a}, E \vdash \texttt{fun }f\texttt{(}x\texttt{) = }e \Downarrow_{\mathrm{p}} [E, \texttt{fun }f\texttt{(}x\texttt{) = }e], \overline{a}}$$

$$\text{app}\frac{\begin{array}{c}\overline{a}, E \vdash e \Downarrow_{\mathrm{p}} [E', \texttt{fun }f\texttt{(}x\texttt{) = }e''], \overline{a}' \qquad \overline{a}', E \vdash e' \Downarrow_{\mathrm{p}} v', \overline{a}'' \\ \overline{a}'', E'[f \mapsto [] * [E', \texttt{fun }f\texttt{(}x\texttt{) = }e''], x \mapsto [] * v'] \Downarrow_{\mathrm{p}} v, \overline{a}'''\end{array}}{\overline{a}, E \vdash e\ e' \Downarrow_{\mathrm{p}} v, \overline{a}'''}$$

$$\text{let}\frac{\overline{a}, E \vdash e \Downarrow_{\mathrm{p}} v', \overline{a}' \qquad \overline{a}', E[x \mapsto [] * v'] \vdash e' \Downarrow_{\mathrm{p}} v, \overline{a}''}{\overline{a}, E \vdash \texttt{let }x\texttt{ = }e\texttt{ in }e' \Downarrow_{\mathrm{p}} v, \overline{a}''}$$

$$\text{letpair}\frac{\overline{a}, E \vdash e \Downarrow_{\mathrm{p}} \texttt{(}v_p\texttt{, }v_p'\texttt{)}, \overline{a}' \qquad \overline{a}', E[x \mapsto v_p, x' \mapsto v_p'] \vdash e' \Downarrow_{\mathrm{p}} v, \overline{a}''}{\overline{a}, E \vdash \texttt{let (}x\texttt{, }x'\texttt{) = }e\texttt{ in }e' \Downarrow_{\mathrm{p}} v, \overline{a}''}$$

$$\text{letabst}\frac{\begin{array}{c}\overline{a}, E \vdash e \Downarrow_{\mathrm{p}} \texttt{<<}a\texttt{>>}v_p', \overline{a}' \\ \overline{a}' \uplus \{a'\}, E[x \mapsto [] * a', x' \mapsto (a\ a') :: [] * v_p'] \vdash e' \Downarrow_{\mathrm{p}} v, \overline{a}''\end{array}}{\overline{a}, E \vdash \texttt{let <<}x\texttt{>>}x'\texttt{ = }e\texttt{ in }e' \Downarrow_{\mathrm{p}} v, \overline{a}''}\ a' \in \mathbb{A} \setminus \overline{a}'$$

$$\text{if}\frac{\begin{array}{cc}\overline{a}, E \vdash e \Downarrow_{\mathrm{p}} a, \overline{a}' & \overline{a}', E \vdash e' \Downarrow_{\mathrm{p}} a', \overline{a}'' \\ a = a' \Rightarrow \overline{a}'', E \vdash e'' \Downarrow_{\mathrm{p}} v, \overline{a}''' & a \neq a' \Rightarrow \overline{a}'', E \vdash e''' \Downarrow_{\mathrm{p}} v, \overline{a}'''\end{array}}{\overline{a}, E \vdash \texttt{if }e\texttt{ = }e'\texttt{ then }e''\texttt{ else }e''' \Downarrow_{\mathrm{p}} v, \overline{a}'''}$$

$$\text{match}\frac{\exists j.\ (\overline{a}, E \vdash e \Downarrow_{\mathrm{p}} \mathtt{C}_j\texttt{(}v_p\texttt{)}, \overline{a}' \ \wedge\ \overline{a}', E[x_j \mapsto v_p] \vdash e_j \Downarrow_{\mathrm{p}} v, \overline{a}'')}{\overline{a}, E \vdash \texttt{match }e\texttt{ with }\cdots\texttt{| }\mathtt{C}_k\texttt{(}x_k\texttt{) -> }e_k\texttt{ | }\cdots \Downarrow_{\mathrm{p}} v, \overline{a}''}$$

Figure 7.1: Big-step semantics with delayed permutations.

Conjecture 7.1.1. *Given any environment E, write $e[E]$ for that expression formed by substituting x for $E(x)$ throughout e, for each $x \in \mathrm{dom}(E)$. Then for any Mini-FreshML evaluation judgement $\overline{a}, e[E] \Downarrow v, \overline{a}'$ there exists at least one judgement $\overline{a}, E \vdash e \Downarrow p, \overline{a}'$ with $v = \mathrm{erase}(p)$. Moreover, given a judgement $\overline{a}, E \vdash e \Downarrow_{\mathrm{p}} p, \overline{a}'$ then there exists a unique judgement $\overline{a} \vdash e[E] \Downarrow \mathrm{erase}(p), \overline{a}'$.* $\diamond$

Note that the Proposition is carefully worded to take into account the fact that $\mathrm{erase}(-)$ is not injective: there may exist multiple values with delayed permutations which map onto the same canonical form. (Take for example atoms a, a' and consider the two values $[] \cdot a$ and $((a\ a') \cdot ::[]) \cdot a'$.)

7.1.2 Data structures and algorithms

Within the Objective Caml standard library are many useful modules implementing standard data structures and their associated algorithms. Here we shall single out just two: hash tables

and finite maps (the latter being implemented using balanced binary trees). These are worthy of study due to the issues which arise when attempting to store atoms within them. This is a very common occurrence: for example, we often wish to represent an environment inside an evaluator using a map from atoms to values.

Unfortunately, the interactions between such data structures and our freshness features are non-trivial to say the least. In particular, there are two serious problems which arise:

1. There is no ordering on atoms, which inhibits the writing of comparison functions (which are needed in order to use the `Map.Make` functor, for example).

2. Data structures such as hash tables and balanced binary trees are not equivariant constructions, in some sense: blindly swapping atoms throughout them destroys their invariants. In the case of a hash table keyed on atoms, for example, a swapping operation may require values to switch buckets (in the case where the transpositions affect the keys in such a way as to make them correspond to alternative buckets).

A corollary of point 1 above is that there is no immediate way to 'hash atoms' (say for calculating hash table bucket numbers).

Recall (§6.3) that atoms inside the runtime system are represented by integers. This can actually be exposed at the language level[1] through the use of the function `Hashtbl.hash`. (For example, this could be used as the key for a balanced binary tree mapping from atoms to pieces of data.) We conjecture that the language still correctly represents α-equivalence classes of object language terms even in the presence of an atom-hashing primitive (because it cannot be used to bypass the procedure for deconstructing abstraction values), although this is still not entirely clear and it remains an item of future work to rigorously prove it.

Now what about the problem of fundamentally non-equivariant constructions? It appears that outsourcing some of the swapping operations to the user-level may help here. Basically, the idea is to enable the programmer of one of the offending modules to specify a function which should be called by the runtime system whenever a swapping operation must be performed upon such a data structure. The programmer would have to ensure that such a function satisfied the necessary algebraic identities (viz. those in §4.1). Recent work by Sewell, Wansbrough et al. on the Acute project[57], which has produced around 25,000 lines of Fresh O'Caml code, seems to uphold our belief that this proposal would be useful. Using the hash table example, the programmer could specify a function which ensures that values are moved between buckets if the swapping affects the keys of the table. Similar transformations could be effected if balanced binary trees are keyed on atoms. Depending on the particular data structure in question, the programmer may wish to delay swapping operations until they are really needed in order to improve efficiency.

At the current time, there is some very experimental support (through an undocumented language construct known as `custom_swap`) for this user-level swapping. It is likely that this will be made robust in the near future. Hopefully, this will make the O'Caml standard library much more useful for Fresh O'Caml programmers.

7.1.3 Objects and modules

Being fundamental parts of the O'Caml language and important tools for structuring programs, we must say a few words about objects and modules. How do such features interact with our facilities for handling binding? In the case of objects, the interactions have not been studied: it is an item of future work which would be worth pursuing. As for modules, we believe that the system is well-behaved, although no formal correctness results have been proven. It is however certainly the case that swapping may occur on values of abstract type with no difficulties, because the values really are concrete when seen by the runtime system.

7.1.4 Reference cells

We saw in §2.6 that the addresses of reference cells are treated as being pure. A major item of future work is to develop the necessary theory to change this view of references, which comes down to to formulating a suitable notion of swapping on reference cells. Naïve solutions do not

[1] Well, in fact they can be exposed using functions inside the `Obj` module as well. However, such operations are naughty and enable you to break pretty much any safety property...

▷ *Fragment 1:*

```
let a = fresh in
let r = ref a in
let x = <<a>>r in
  match x with <<p>>q -> !r = a;;
```

▷ *Fragment 2:*

```
let a = fresh in
let x = <<a>>(ref a) in
  match x with <<p>>q ->
    match x with <<p'>>q' -> !p' = !q';;
```

Figure 7.2: Examples involving mutable state.

work: if the result of swapping atoms a, a' on a reference cell containing a were to be the same reference cell with the contents changed to a', then highly unpredictable behaviour would result. Imagine for a few moments that we use this semantics and consider the fragments in Figure 7.2. The result of evaluating both code fragments would in fact be `false`. In the first, the pattern-match causes side-effects not only in the generation of the fresh name to be bound to `p`, but also in updating the reference cell. The second example is even more pertinent: after the first match, the value `x` will be of the form `<<`a`>>ref` a' for atoms a, a'—the binding structure has been destroyed. The second match enables us to identify this.

We believe that a sensible approach might view references as presenting particular 'views' onto graph structures. Whilst a notion of α-equivalence on such structures should not be problematic to define, there are difficulties arising from the runtime pointer-chasing which this could involve.

Discussions with Xavier Leroy identified a possible solution which would exploit the delayed permutations discussed in §7.1.1. The idea is that transpositions may be applied to reference addresses, but remain delayed there until the cell is dereferenced. At that time, a copy of the reference's contents will be returned with the pending permutation now attached to them. This provides multiple 'views' onto the graph structure, each of which may be equipped with different permutations. One difficulty with this scheme, however, is working out how to formulate a suitable notion of equality on references. At first sight, it seems as if the way to compare two reference addresses for equality would be to proceed as normal, discarding any pending permutations. However, then the situation could arise where two reference addresses are equal but their contents differ (due to different pending permutations being attached to each address, which are to be applied only at dereference-time).

Future work must establish whether a sound scheme based on this proposal is viable and how difficult it would be to implement.

7.1.5 Enhanced abstraction types

Another fruit of the work on the Acute system[57] has been some ideas relating to enhanced abstraction types. The first proposal is to introduce a distinguished type constructor, perhaps called `nobind`, which identifies points in syntax trees under which the transpositions introduced by deconstructing abstraction values would be nullified. (One could also consider having the opposite situation, where abstraction expressions do not by default represent binding constructs. To indicate an occurrence of a bound name in the syntax tree, one would have to tag the corresponding part of the `type` declaration with a distinguished constructor `bind`.) To see how all of this works, consider a grammar for patterns which can contain both pattern variables and type variables (the latter encapsulated inside explicit type annotations). Both of these varieties of variable would likely be represented in Fresh O'Caml by different types of bindable names. The patterns could then occur within a syntax tree representing a `match` construct, say, which only binds pattern variables and not type variables. The new scheme permits an encoding as follows.

```
type t and var = t name;;
type t and tyvar = t name;;
type ty = Tvar of tyvar
        | ... ;;
type pat = Pvar of var
         | Ptyped of pat * (ty nobind)
         | ... ;;
type expr = Ematch of expr * ((<<pat>>expr) list)
          | ... ;;
```

Now one can have only the pattern variables freshened up when deconstructing a data value built with `Ematch`.

A second proposal relates to improvements in efficiency. In a large-scale system, it is clear (and has been experimentally verified) that the overhead of atom-swapping throughout syntax trees is significant. Even if the method of delayed permutations (§7.1.1) were to be implemented, it would still be highly desirable to minimise the amount of swapping required. One means of doing this would be to allow the programmer to specify that swapping operations should never look inside certain parts of values.

The particular example cited by Wansbrough (private communication) concerns a type declaration for abstract syntax trees containing closures. Such a closure is designed to always have empty support (since it encapsulates all of the free variables of the relevant function within it). Therefore no swapping operation ever needs to propagate inside. However, there is currently no way of preventing this. It is thought that a distinguished type constructor could be used to identify parts of values which are to be bypassed in this way. The `nobind` constructor from above may suffice for this purpose. Of course, it is then up to the programmer to ensure that the special tag is used sensibly—rather than the language ensuring safety—but we do not believe that incorporating such a feature would be dangerous (in the sense of directly compromising the correctness properties of Chapter 5).

7.1.6 Standard library enhancements

There is much to be said for minimising the number of language primitives which are hard-coded into the Fresh O'Caml compiler. This can be done by moving functionality into the standard library, making it easier to migrate the compiler modifications to new versions of the O'Caml system. It is not quite clear where the divide between inbuilt and library functionality should lie, however. One reasonable scenario would be to keep the expressions for generating fresh names together with those for constructing and deconstructing abstractions as inbuilt features. The `swap` and `freshfor` keywords would then be replaced by functions in the standard library.

Alongside these modifications, additional library functions could usefully be provided. A function permitting the application of non-trivial permutations of atoms (rather than just single swaps) to values could be useful; meanwhile, both this and the normal swapping function could be specialised to variants which provide 'fresh renaming' as a convenience.

It might also be useful to incorporate a function which takes an arbitrary value and returns its algebraic support. (Such a function already exists inside the runtime system, but is not available to the programmer.) However, it is not clear what type such a function could be assigned: the atoms throughout the value may be of different bindable types. It is possible that a more advanced sorting system for atoms (viz. §2.1.1) would help here.

7.1.7 Denotational semantics

It is clear that our denotational semantics could be of use in order to prove further properties of Mini-FreshML-like languages. For example, it may be possible to prove Conjecture 7.1.1 via denotational methods. The denotational semantics could be extended to provide reasoning principles about other language constructs which might be present in Fresh O'Caml. One obvious place where the semantics is lacking is in support for abstraction types more complicated than just `<<name>>`τ. Fresh O'Caml supports such types and provides algorithms for computing equality between values of such types, but this does not currently have a justification in our denotational semantics. In order to support this, FM-domain theory needs to be further developed to allow FM-cppos $[D]D'$ to be constructed for arbitrary D and D'.

Whilst one can sketch possible constructions of such an FM-cppo (whose counterparts in FM-*set* theory are well-understood) it is not at all clear whether, for example, they are suitably chain-complete; previous experiences with dynamic allocation monads (§5.1.1) suggest that such order-theoretic completeness properties may prove troublesome.

It should also be noted that generalised abstraction types $\texttt{<<}\tau\texttt{>>}\tau'$ are very difficult to incorporate with the procedure of freshness inference. For given an identifier `x` and an expression $E \stackrel{\text{def}}{=} \texttt{<<}e\texttt{>>}e'$, where e is of some arbitrary type τ, then we can only deduce $\texttt{x} \mathbin{\#} E$ iff the identifier `x` is *definitely not* fresh for e, or else if `x` is certainly fresh for e and also e'. The difficulty lies in the 'not fresh' part, for this necessitates building up sound judgements[2] $x \neg\# e$ as well as the 'fresh for' judgements. Despite some (unpublished) work on the subject by the author, no satisfactory algorithm has yet been found to infer such judgements.

Another interesting thing to note is that we should not necessarily restrict ourselves to using the continuation monad $(D \multimap R) \multimap R$. There are others, for example:

$(D \to R) \multimap R$	non-strict use of the continuation's argument
$(D \multimap R) \to R$	non-strict use of the continuation

The choice of which monad to use should be driven by the operational semantics of the language. For example, a language which permits any form of control flow enabling a program to abort without examining the current continuation would best be modelled by the $(D \multimap R) \to R$ monad. And although we have not investigated it further, we also note that our current denotational semantics is an example of *linearly-used* continuations[7].

A useful extension to the denotational semantics would be a treatment of imperative features such as references: mutable state can be easily captured by using $S \multimap 1_\perp$ instead of just $1_\perp$ throughout the semantics, given a recursively-defined FM-cppo S of *stores*. Indeed, recent work by Benton et al.[6] does indeed develop a theory along these lines with a view to verifying equivalences such as the famous Meyer-Sieber examples[32].

[2] 'Staleness judgements' might be a suitable name for such statements, although that name has not found favour with some!

Bibliography

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lèvy. Explicit substitutions. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*, pages 31–46. ACM, 1990.

[2] S. Abramsky, D. Ghica, A. Murawski, L. Ong, and I. D. B. Stark. Nominal games and full abstraction for the nu-calculus. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science*, pages 150–159. IEEE Computer Society Press, 2004.

[3] S. Abramsky and A. Jung. Domain theory. In *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Clarendon Press, 1994.

[4] D. Ancona and E. Moggi. A fresh calculus for name management. In *Generative Programming and Component Engineering*, volume 3286 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[5] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.

[6] N. Benton and B. Leperchey. Relational reasoning in a nominal semantics for storage (preliminary report). Available from `http://research.microsoft.com/~nick/`, November 2004.

[7] J. Berdine, P. O'Hearn, U. Reddy, and H. Thielecke. Linear continuation-passing. *Higher Order and Symbolic Computation*, 15(2/3):181–208, 2002.

[8] R. M. Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12(1):41–48, 1969.

[9] L. Cardelli, P. Gardner, and G. Ghelli. Manipulating trees with hidden labels. In A. D. Gordon, editor, *Proceedings of the 6th International Conference on Foundations of Software Science and Computation Structures (FOSSACS 2003)*, volume 2620 of *Lecture Notes in Computer Science*, pages 216–232. Springer-Verlag, 2003.

[10] 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)*, volume 3132 of *Lecture Notes in Computer Science*, pages 269–283. Springer-Verlag, 2004.

[11] N. de Bruijn. Lambda calculus notation with nameless dummies: a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–391, 1972.

[12] N. de Bruijn. A survey of the project AUTOMATH. *To H. B. Curry: essays on combinatory logic, lambda calculus, and formalism*, pages 597–606, 1980.

[13] D. de Rauglaudre. Camlp4 reference manual. Available from `http://caml.inria.fr/camlp4/index.html`, 2004.

[14] J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-order abstract syntax in Coq. In M. Dezani and G. Plotkin, editors, *Proceedings of the TLCA'95 International Conference on Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 124–138. Springer-Verlag, 1995.

[15] J. Despeyroux, F. Pfenning, and C. Schürmann. Primitive recursion for higher-order abstract syntax. In P. de Groote and J. R. Hindley, editors, *Proceedings of the TLCA'97 International Conference on Typed Lambda Calculi and Applications*, volume 1210 of *Lecture Notes in Computer Science*, pages 147–163. Springer-Verlag, 1997.

[16] M. Fernández, M. Gabbay, and I. Mackie. Nominal rewriting. Submitted, January 2004.

[17] M. P. Fiore, G. D. Plotkin, and D. Turi. Abstract syntax and variable binding. pages 193–202, 1999.

[18] M. J. Gabbay. *A Theory of Inductive Definitions with α-Equivalence: Semantics, Implementation, Programming Language*. PhD thesis, University of Cambridge, 2000.

[19] M. J. Gabbay. Automating Fraenkel-Mostowski syntax. In *15^{th} International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, 2002. Work-in-progress submission, to be published as a NASA technical report.

[20] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2002.

[21] M. J. Gabbay and L. Wischik. Implementing a bisimulation checker in Fresh O'Caml. Submitted, April 2004.

[22] J. Y. Girard. *Interprétation functionelle et élimination des coupures dans l'arithmétique d'ordre supérieure*. PhD thesis, Université Paris VII, 1972.

[23] J. Y. Girard. The system F of variable types, fifteen years later. *Theoretical Computer Science*, 45:159–192, 1986.

[24] M. Hofmann. Semantical analysis of higher-order abstract syntax. In *14^{th} Logic in Computer Science Conference (LICS'99)*, pages 204–213. IEEE Computer Society Press, 1999.

[25] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In L. Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146. Addison-Wesley, 1999.

[26] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, 1988.

[27] F. Le Fessant and L. Maranget. Optimizing pattern-matching. In *Proceedings of the 2001 International Conference on Functional Programming (ICFP'01)*, pages 26–37. ACM Press, 2001.

[28] P. B. Levy. Martin-Löf Clashes With Griffin, Operationally. Available from `http://www.cs.bham.ac.uk/~pbl/papers/`.

[29] I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):287–327, 1991.

[30] J. McCarthy. Towards a mathematical science of computation. *IFIP Congress 1962*, 1963.

[31] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes, and barbed wire. In *Proceedings of the FPCA'91 conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, 1991.

[32] A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables. In *Proceedings of the 15^{th} ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages, POPL'88*, pages 191–203. ACM Press, 1988.

[33] D. A. Miller. An extension to ML to handle bound variables in data structures: Preliminary report. In *Proceedings of the Logical Frameworks BRA Workshop*, 1990.

[34] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

[35] R. Milner. What's in a name? In A. Herbert and K. Spärck Jones, editors, *Computer Systems: Theory, Technology and Applications*. Springer-Verlag, 2003.

[36] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML: Revised*. The MIT Press, 1997.

[37] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Department of Computer Science, University of Edinburgh, 1989.

[38] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[39] R. M. Needham. Names. In S. Mullender, editor, *Distributed Systems*, pages 315–326. Addison-Wesley, 1993.

[40] S. L. Peyton Jones. Tackling the awkward squad: Monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In C. A. R. Hoare, M. Broy, and R. Steinbruggen, editors, *Engineering Theories of Software Construction*, pages 47–96. IOS Press, 2001.

[41] S. L. Peyton Jones. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4):393–434, 2002.

[42] S. L. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003. Also available from `http://www.haskell.org/definition/`.

[43] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI)*, pages 199–208. ACM Press, 1988.

[44] A. M. Pitts. Typed operational reasoning. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 3, pages 165–199. The MIT Press. To appear.

[45] A. M. Pitts. Computational adequacy via 'mixed' inductive definitions. In *Proceedings of the 9th International Conference on Mathematical Foundations of Programming Semantics*, volume 802 of *Lecture Notes in Computer Science*, pages 72–82. Springer-Verlag, 1994.

[46] A. M. Pitts. Relational properties of domains. *Information and Computation*, 127:66–90, 1996. (A preliminary version of this work appeared as Cambridge University Computer Laboratory Technical Report 321, December 1993).

[47] A. M. Pitts. A note on logical relations between semantics and syntax. *Logic Journal of the Interest Group in Pure and Applied Logics*, 5(4):589–601, July 1997.

[48] A. M. Pitts. Operational semantics and program equivalence. In G. Barthe, P. Dybjer, and J. Saraiva, editors, *Applied Semantics, Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science, Tutorial*, pages 378–412. Springer-Verlag, 2002.

[49] A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Proceedings of the 5th International Conference on Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, 2000.

[50] A. M. Pitts and I. D. B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In *Mathematical Foundations of Computer Science, Proc. 18th Int. Symp., Gdańsk, 1993*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer-Verlag, 1993.

[51] A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 227–273. Cambridge University Press, 1998.

[52] G. D. Plotkin. Pisa notes (on domain theory). Available from `http://homepages.inf.ed.ac.uk/gdp/publications/`, 1983.

[53] C. Raffalli. A package for abstract syntax with binder. 1998.

[54] J. C. Reynolds. Towards a theory of type structure. In *Paris Colloquium on Programming*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.

[55] J. C. Reynolds. Types, abstraction, and parametric polymorphism. *Information Processing*, pages 513–523, 1983.

[56] U. Schöpp and I. D. B. Stark. A dependent type theory with names and binding. In *Computer Science Logic: Proceedings of the 18th International Workshop CSL 2004*, volume 3210 of *Lecture Notes in Computer Science*, pages 235–249. Springer-Verlag, 2004.

[57] P. E. Sewell, J. J. Leifer, K. Wansbrough, M. Allen-Williams, F. Zappa Nardelli, P. Habouzit, and V. Vafeiadis. Acute: high-level programming language design for distributed computation. Design rationale and language definition. University of Cambridge Computer Laboratory Technical Report 605. Available from `http://www.cl.cam.ac.uk/users/pes20/acute/index.html`, 2004.

[58] T. Sheard. Accomplishments and research challenges in meta-programming. In volume 2196 of *Lecture Notes in Computer Science*, Springer-Verlag, 2001.

[59] M. R. Shinwell. The implementation of FreshML. Dissertation for Part II of the Cambridge University Computer Science Tripos, 2000. Available from `http://www.cl.cam.ac.uk/~mrs30/`.

[60] M. R. Shinwell. Swapping the atom: Programming with binders in Fresh O'Caml. Proceedings of the MERλIN Workshop, 2003.

[61] M. R. Shinwell and A. M. Pitts. On a monadic semantics for freshness. *Theoretical Computer Science*. To appear; a preliminary version appears in the proceedings of the second workshop of the EU FP5 IST thematic network IST-2001-38957 APPSEM II, Tallinn, Estonia, April 2004.

[62] M. R. Shinwell and A. M. Pitts. *Fresh O'Caml User Manual*. Cambridge University Computer Laboratory, 2003. Available at `http://www.freshml.org/`.

[63] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*, pages 263–274. ACM Press, 2003.

[64] M. B. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal of Computing*, 11(4):761–783, 1982.

[65] I. D. B. Stark. *Names and Higher-Order Functions*. PhD thesis, University of Cambridge, December 1994. Also available as University of Cambridge Computer Laboratory Technical Report 363.

[66] I. D. B. Stark. Categorical models for local names. *Lisp and Symbolic Computation*, 9(1):77–107, 1996.

[67] C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. In M. Baaz, editor, *Computer Science Logic and 8th Kurt Gödel Colloquium (CSL'03 & KGC), Vienna, Austria.*, volume 2803 of *Lecture Notes in Computer Science*, pages 513–527. Springer-Verlag, 2003.

[68] P. Wadler. Compilation of pattern matching. In S. L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, chapter 7. Prentice-Hall International, 1987.

[69] P. Wadler. Theorems for free! In *4th International Symposium on Functional Programming Languages and Computer Architecture*. ACM Press, 1989.

[70] G. Washburn and S. Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*, pages 249–262. ACM Press, 2003.

[71] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.