# STRICTNESS ANALYSIS APPLIED TO PROGRAMS WITH LAZY LIST CONSTRUCTORS

Cordelia V. Hall

Submitted to the faculty of the Graduate School
in partial fulfillment of the requirements
of the degree
Doctor of Philosophy
Indiana University
Department of Computer Science

December, 1987

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements of the degree of Doctor of Philosophy.

David S. Wise, Ph.D.

Daniel P. Friedman

Doctoral Committee

Steven D. Johnson

Vernon Kliewer

Paul W. Purdom

September 4, 1987

## Acknowledgements

Several people have given me their time, interest and support (financial and otherwise) during the process of completing this work.

My thesis advisor, David Wise, provided an excellent working environment which was both challenging and unusually supportive; it allowed me to experiment and grow. He also helped me finance my education with money from an NSF research grant for several years. Steve Johnson acted as an advisor during Wise's sabbatical year, and afterwards continued to provide interesting and useful discussions as well as tactful support when necessary. I am also grateful to my committee members, Paul Purdom and Dan Friedman, for the interest they showed in my work, and to Vernon Kliewer for acting as my outside committee member.

I learned, and continue to learn, much from John O'Donnell, with whom it is possible to have useful technical discussions with a minimum of explanation. His support and interest have been invaluable to me, as has that of the rest of my family.

## Abstract

Strictness analysis applied to programs with lazy list constructors

Cordelia V. Hall

*Lazy* applicative languages are more powerful than conventional languages, in the sense that they can avoid an unnecessary infinite loop by delaying the computation of an argument, computing its value only after determining that it is essential to the behavior of the program. However, the mechanism used to delay such evaluations requires time and space, which may be wasted if the program eventually needs the values anyway. This dissertation demonstrates how a compiler for a lazy applicative language can identify many arguments that can be evaluated early, while still avoiding any premature computation of values that might cause a program to loop.

This work has a practical goal: to use strictness analysis to annotate code so that it retains its original semantics, but runs more efficiently. Unlike other efforts to solve this problem, both non-flat data and function domains are considered. The essential tool is analysis of list/record construction; such operations are readily recognizable from syntax, and occur every time a list of arguments is passed to a function. In practice, the strictness in fields within those records often follows regular patterns that can be finitely represented. Of particular interest are programs that manipulate useful structures such as *streams*. When compiled using the approach presented here, these programs typically contain a small number of efficient, mutually recursive loops, causing the exchange of a small increase in overall code size for a large decrease in space and time consumption as the stream is produced.

Weak and strong safety become important issues and are discussed at length. Termination is guaranteed by several factors, including a finite resource which controls the increase in code size, and a regularity constraint placed upon the strictness patterns propagated during compilation. The compiler is proved to be safe relative to an axiomatic specification of an interpreter. Limited extension of the analysis through conditional expressions and to higher-order functions is possible.

## Table of Contents

# Chapter 1: Introduction

This work presents a source-to-source compiler intended to improve the time and space behavior of programs written in a class of higher order, statically scoped languages referred to here as *applicative lazy* languages. These languages are lazy (all expressions are evaluated at most once), and they are applicative, meaning that they have no side-effects. A lazy evaluator was simultaneously described by Henderson and Morris [14] and Friedman and Wise [11], who produced an interpreter with equivalent semantics by implementing a traditional Lisp interpreter with lazy **cons**. The lazy source language treated here is Daisy, a descendant of the interpreter presented by Friedman and Wise.

Friedman and Wise assert that **cons** should not evaluate its arguments [11]. This approach is elegant and powerful, but expensive. The thesis of this work is to show *how to safely compile special cases in which* **cons** *should evaluate its arguments.*

## Section 1.1: Applicative lazy languages

Applicative lazy languages, such as SASL[36], LML[2], Ponder[10], or Daisy[20], have many properties worth exploring. They have no side-effects, a fact which makes them interesting candidates for general-purpose parallel programming languages because control-flow problems are removed, leaving only the problem of reducing data dependencies. They produce values where applicative order, or call-by-value, languages loop forever. Finally, they permit expressions to be substituted for equivalent expressions, providing programs which are easier to reason about, thus supporting automatic proofs of correctness.

Unfortunately, implementations of these languages tend to be slow. "Lazy" or "delayed" evaluation provides the semantic power of these languages by permitting any given computation to avoid calculating values which are not required in computing the final value. This is generally implemented by a mechanism similar to Algol's *call-by-name*, except that instead of computing the value each time

it is required (necessary in a language with side-effects), the value is comp only once. Usually, this mechanism, referred to here as a *suspension*, is plemented in a general way that does not distinguish between values that eventually be required and values that are never needed.

Suspensions are expensive. They can be regarded as process control bl that contain a bit indicating whether the delayed expression has been evalua and if not, a pointer to the environment current during runtime when the pension was created and a pointer to the code whose evaluation will produc value represented by this suspension. Each time a value is required from a pension, that bit must be tested before the extant value can be used (Bloss Hudak [3] develop techniques for the detection of unnecessary tests). If it not exist, an expensive context swap is necessary. Environments must be served until no suspensions point to them. The costs of suspensions are incre when the expression being suspended itself depends on a suspension.

Suspensions representing values which *will* be required (with some inte ing restrictions) needn't be suspensions at all — the values they represent as well be computed at once, because they are going to be computed any If they can be computed at once, then all of the overhead involved in cre a suspension is avoided. This is the essential idea that has recently persu many people to examine *strictness analysis*.

## Section 1.2: Strictness analysis

Strictness analysis calculates the relationship between a function's a ments and its result. A function is said to be *strict* in argument $n$ if the fun is $\perp$ when argument $n$ is $\perp$, where $\perp$ is the undefined element that gene represents an infinite loop in the domain of S expression values. For exa binary addition must use both addends to compute its result; it is strict in arguments when its result is required. However, a conditional expression req the value of its first argument in order to determine which of its other argu will be required to produce a result. In general, it is possible to determine v

argument will be required only at run-time; this makes strictness an undecidable problem for compilers.

The relationship between strictness analysis and the safe removal of suspensions is simple. If a function is strict in an argument $n$, and if the function's result will be required by the whole computation, then there is no point in suspending either $n$ or the function's result. Moreover, there is no point in suspending values upon which the computation of the value of $n$ depends. Since the whole computation would loop if any of these values looped, there is no harm in letting a loop occur at a different point in the computation than it would have if the computation had been completely lazy. Of course, if none of these values looped, then there is no point in suspending them anyway.

The identification of expressions that need not be suspended has some benefits to the use of lazy languages on parallel architectures. Functions, such as **add**, may be strict in more than one argument. Since there are no side-effects in the language and the values of the arguments are known to be necessary to the computation of the final result, these arguments may be evaluated simultaneously by processors which can be fully committed to their evaluation.

### Section 1.3: Strictness analysis and lists

Many lazy languages are descendants of Lisp [27], and list processing is central to the programming techniques developed by users of these languages. Strictness analysis produces some particularly interesting results when used to compile lazy list-constructing programs, due to the variety of ways in which a list-valued function's result may be used by a calling function. Unlike the result of an addition, which is either required or not required by the caller, a function as simple as **cons** may require neither of its arguments, its first argument, its second argument or both arguments, depending upon the use made of its result. The following four programs (all assumed to be entire programs, executed at top-level and written in Lisp) illustrate these different uses of the result of an application of **Cons**, defined as `(lambda (a b) (cons a b))`:

```
— ((lambda (a b) b) (Cons 1 2) 3)
— ((lambda (a) (car a)) (Cons 4 5))
— ((lambda (a) (cdr a)) (Cons 6 7))
— ((lambda (a) (add (car a) (cdr a))) (Cons 8 9))
```

Consider a program in which all four of these expressions appeared and produced values necessary to the final value, such as:

```
(add
  (add ((lambda (a b) b) (Cons 1 2) 3)
       ((lambda (a) (car a)) (Cons 4 5)))
  (add ((lambda (a) (cdr a)) (Cons 6 7))
       ((lambda (a) (add (car a) (cdr a))) (Cons 8 9))))
```

Suppose that such a program is to be compiled so that the arguments to **Cons** are evaluated immediately wherever possible, and arguments in which **Cons** is strict are annotated. There are two choices in compiling the body of **Cons**:

1) **Cons** can be compiled once as if it was strict in neither of its arguments. This approach has the advantage of keeping the code size small, but the disadvantage of ignoring the strictness information that can be discovered at compile-time. If the compiler marked (with $) all of the expressions that it could safely determine to be evaluated early, it would produce the following code:

```
(add
  $(add $((lambda (a b) b) (Cons 1 2) $3)
      $((lambda (a) (car a)) (Cons $4 5)))
  $(add $((lambda (a) (cdr a)) (Cons 6 $7))
      $((lambda (a) (add $(car a) $(cdr a))) (Cons $8 $9))))
```

(add's arguments are grouped by an implicit *cons*).

However, **Cons**, the function `(lambda (a b) (cons a b))`, would remain lazy, meaning that, wherever called, it creates unnecessary suspensions. The number of these suspensions is trivial here, but becomes increasingly important as function definitions grow larger, are executed often and appear in many parts of the program.

2) **Cons** can be compiled in as many different ways as there are different uses of its result. These are the four possible *versions* of **Cons** [6,9, 41]:

```
— (lambda (a b) (cons a b))
— (lambda (a b) (cons $a b))
— (lambda (a b) (cons a $b))
— (lambda (a b) (cons $a $b))
```

The use of **Cons** versions has the advantage of making full use of the information that can be determined at compile time, but the disadvantage of producing four definitions of **Cons**, three more than would otherwise be needed. This is a disadvantage when **Cons** is not executed frequently, but the advantages of versions greatly increase when a version of **Cons** is often executed.

### 1.3.1: Lists with looping components

Looping programs may produce values that are not $\perp$, but that contain $\perp$ as a component. For example, a program may produce the partial list `(1,2,...` before looping. This would be represented as $\langle 1, \langle 2, \perp \rangle \rangle$, which is distinct from

$\perp$. Strictness introduced by compilation could cause a looping element, which would ordinarily have been evaluated only after the 1 and 2, to be evaluated before either of the other values, so that the complete value of the interpretation of the source code would no longer be equal to the value of the interpretation of the object code, even though *eventually* both values would loop. In such a case, the assumption that makes strictness analysis an interesting aid in designing a compiler—that looping expressions may safely be evaluated at any time during a program's execution if they can be shown to be eventually required by the computation—needs to be examined more carefully.

In fact, if the interpreter is assumed to be truly lazy, then the left parenthesis of a list value is printed before any attempt is made to determine the first element of the list, since it doesn't need to know the value of the head of the list to determine that it is printing a list and that a left parenthesis is needed. Any analysis that shows that the first element of the list will be needed and that asserts that this information justifies the earlier evaluation of this element may cause the interpretation of the compiled code to loop without printing anything—not even a left parenthesis. Unfortunately, the problem of determining when the compiler is justified in marking this first element is in general undecidable.

Two kinds of safety may be defined (and were proposed by O'Donnell [33]):

1) Weak safety means that the interpretation of source and compiled code is equal when the interpretation of the source code is not $\perp$ and does not contain $\perp$.
2) Strong safety means that the interpretation of source and compiled code produce the same element in the lattice of values.

It is also possible to weaken the definition of strong safety so that programs may be proven equal if their results printed on a terminal screen are identical on a non-null prefix, which may be infinite. These definitions will be discussed in more detail in Chapter 2.

### 1.3.2: Infinite lists

Infinite lists can represent objects such as a stream of electrical impulses in a circuit design [19], or the infinite dialogue between user and machine that is the heart of a programming environment [13,30] or operating system [32]. They have also been useful in developing hardware description languages [31].

Infinite lists present special problems to a compiler designer bent upon using strictness analysis to improve lazy programs. Unfortunately, it isn't possible both to preserve the semantics of programs that produce them and to annotate every element of the list which may eventually be required because the order in which these list elements are produced is very significant — the user expects to see the initial elements before seeing elements farther down in the list, and certainly expects to see them before the end of the list appears. The basic assumption driving the application of strictness analysis to applicative lazy programs is violated when applied to infinite lists, because they are not $\perp$, yet their components (specifically their recursively defined tails) cannot be evaluated in any order if an interesting approximation to their complete value is to appear on a terminal screen.

In order to preserve the behavior of infinite lists, the compiler avoids marking code that generates their tails, a technique that will be discussed in more detail in Chapter 2.

## Section 1.4: Brief overview of techniques presented here

### 1.4.1: Daisy — the source and target language

Daisy is an applicative statically-scoped lazy language developed from the original Lisp interpreter. Delayed evaluation is achieved by altering *only* the behavior of **cons**, a few list predicates, and list accessing functions such as **car** and **cdr**. The interpreter itself continues, as does the Lisp interpreter, to use *eager* evaluation, causing all function arguments to be evaluated regardless of whether they are required. By delaying the evaluation of its arguments, **cons** provides

the only laziness required to produce normal order semantics. Functions receive a single argument, which may be a list. Often, the function treats sub-structures of this argument list as if they were distinct arguments; it is this convention which guarantees that evaluation of function arguments (sub-structures) will be delayed until the argument value is required by the computation.

### 1.4.2: Demand generated by the printer

Consider a program such as `(cons 1 2)`. It is clear that both argument values will eventually appear in the resulting value `(1 . 2)`. However, the interpreter, as described, produces only a **cons** cell containing two suspensions. What mechanism causes their evaluation?

It is assumed that an external device (the *printer*) demands the entire value of whatever expression it is given, and that it is this device that produces the final result of the evaluation of any program. From this point on, the interpreter will be referred to as if it was composed with the printer.

### 1.4.3: Compiling Daisy programs with strictness patterns

The compiler presented here, and in a preliminary description of this work [12], attempts to predict at compile time the values that the printer will require at run-time, by propagating abstractions of the printer's demand throughout the compilation of sub-expressions in the code tree. These abstractions are referred to here as strictness patterns. A *stream* is used here to refer to a finite or infinite list; this does not prevent a stream from being a tree (or, in the case of data recursion, a graph.)

The initial demand pattern used here is itself recursively defined, although regular, and can be represented with a finite graph. The compiler is shown to propagate this demand with strictness patterns which are either of finite length or are *rational* (a term proposed by Hughes [18]), in order to guarantee that the compiler terminates, a precaution that was independently taken by Hughes in specifying contexts [18].

In addition to marking **cons** arguments that can be shown to be necessary to the computation of a program, the compiler often unrolls loops. In fact, there is an interesting relationship between the propagation of regular patterns and stream-producing functions, which are typically recursive. A function *version* is a pairing of the code representing the function and the pattern propagated to the compilation of the function's application. As a stream-producing function application is compiled with a regular pattern, the compiler encounters the recursive call within the body of the function with a strictness pattern that may itself be regular, depending upon the relationship between the original pattern and the code forming the function body. If the pattern propagated to the compilation of the recursive call is an unfolding of the original pattern, then the recursive call can be compiled as a call to the new version created by the compilation of the original application. If it is not, the function code and the pattern propagated to the compilation of the original call may often be unfolded together, creating new versions during this process, until the unfolded pattern is finally an unfolding of the original pattern. This whole process produces a finite-state automaton formed by versions which refer to each other in a cycle.

## Section 1.5: Other work on strictness analysis

Interest in strictness analysis has steadily grown since Mycroft [28] first used abstract interpretation to determine strictness for flat domains (programs producing atomic values) in 1980. Recent work has centered upon higher order functions in both the typed and untyped lambda calculus, and first order functions on non-flat domains. This body of work is described briefly here; detailed discussion of especially relevant work appears in Chapter 6.

### 1.5.1: Flat domains

Clack and Peyton-Jones [7,8] provide a useful clarification of Mycroft's work on flat domains, and provide measurements of the degree of parallelism achieved by applications of an algorithm similar to that of Mycroft.

### 1.5.2: Higher order functions

Work on strictness analysis of higher order functions is directed towards a variety of problems. Like Clack and Peyton-Jones, Maurer's work [26] is motivated by an interest in exploiting possible parallelism in functional languages. Maurer extends Mycroft's result to typed higher order functions, by representing strictness information with special lambda expressions. He achieves termination using a cutting mechanism that approximates information that might otherwise be derived from the non-terminating analysis of certain expressions, such as an application of the Y combinator.

Wray, and Hudak and Young have produced algorithms for analyzing higher order functions. Hudak and Young [15] develop an algorithm for performing higher order strictness analysis in the untyped lambda calculus, based upon a set-theoretic description of strictness. Wray [40,41] extends Mycroft's result with an algorithm that annotates strict expressions in lazy higher order combinators.

Kuo and Mishra [23] assert that strictness analysis of programs in the untyped lambda calculus is elementary (however, they make no claims concerning the strictness of composite structures) and that strictness analysis can be shown to be a particular case of type inference for the typed lambda calculus. They describe a practical system that performs both type checking and strictness analysis.

Nielson [29] develops a theory of abstract interpretation for the typed lambda calculus, which is shown to also be suitable for strictness analysis. This work is aimed at constructing a general theory for the analysis of functional programs, since, as Nielson observes, otherwise the correctness of strictness analysis must be independently shown for each functional language.

Burn, Hankin and Abramsky [4] use abstract interpretation to analyze higher order functions in the typed lambda calculus. Abramsky [1] extends these results to polymorphic types, showing that strictness analysis for polymorphic functions can be efficiently reduced to strictness analysis for the typed lambda calculus.

### 1.5.3: First order functions with data structures

This is a particularly interesting problem, and is the focus of the work presented here. Various techniques have strong and weak points.

Wadler [37] is able to determine certain kinds of list strictness, such as strictness in all heads and tails, all tails or just the outer structure of the list. While this approach is useful in analyzing some finite lists, the technique does not handle infinite lists, since some tails must remain lazy.

Kieburtz and Napierala [21] use abstract interpretation to develop total interpretations, which can be used by a compiler without risk of non-termination. A total interpretation is then developed for strictness analysis; however, total interpretations apparently yield less strictness information than that of Wadler [37] on finite lists, and an interpretation presented for unbounded data structures is not total.

Hughes [18] analyzes a first order functional language containing only variables, function applications and case expressions, using a simple domain of contexts, which carry strictness information. He proposes this work as a potential basis for further work on practical strictness analysis. He then presents a theoretical framework for strictness analysis of a slightly more powerful language, using continuations to represent contexts [16]. Wadler and Hughes [38] present another theoretical treatment of contexts as retracts, or projections for analysis of a monomorphic first order language.

Lindstrom [25] proposes a domain that contains typing information as well as strictness information for finite lists. This domain is shown to be useful but does not represent patterns that are internally strict but externally lazy. Infinite lists are described as an open problem.

### 1.5.4: Data structures and higher order functions

Hughes [17] compares two approaches to strictness analysis. One is based upon abstract interpretation, which he calls "forward" analysis, and the other

involves reasoning from information about the strictness of an expression to deduce information about a sub-expression, or "backwards" analysis. He argues that backward analysis is likely to be more efficient than forward analysis and that it can be extended to provide strictness information about lists and higher-order functions in typed languages.

### Section 1.6: Outline

Chapter 2 presents a lattice of strictness patterns, and then demonstrates the use of these patterns in a series of equations, with short examples, defining a simple recursive descent compiler. A restriction is placed upon the expressive power of certain propagated cyclic patterns in the compiler equations, which permits the compiler to handle streams. Three more complex examples follow, demonstrating the advantages of version cycles. The compiler is shown to terminate and to propagate only rational patterns.

Chapter 3 proves that the compiler is safe by demonstrating that the strictness patterns propagated by the compiler are always below or equal to those *displayed* by a suitably modified interpreter. This interpreter is identical to the Daisy interpreter, except that it displays a pattern representing the demand of the printer upon the value of the expression currently being interpreted.

The equation compiling conditional expressions in Chapter 2 is expanded in Chapter 4 to handle conditional expressions written in *iterative style* and with *nil?* tests — the chapter includes examples of its application to some well-known functions.

Chapter 5 presents an extension of the compiler discussed in Chapter 2 to higher order functions.

Chapter 6 compares these results to recent results of other researchers in the area, summarizes the contribution of this work, and suggests some interesting possibilities for future study.

# Chapter 2: Compiling strictness into streams

## Section 2.1: Inherited and synthesized strictness patterns

Strictness patterns have two roles. When the strictness pattern is determined from the enclosing evaluation context (such as the exhaustive evaluation of a program's result), it is an *inherited* pattern. Strictness patterns can also be created during the analysis of a function application. When `a` is bound using a $\lambda$-form, `a` may occur several times in the form's body and inherit several patterns, possibly pieces of the pattern inherited by the application. Some combination of these patterns forms a *synthesized* pattern, which becomes the pattern inherited by the argument. For example, if the entire program to be evaluated is `((lambda (a) (head (cons a [])))(cons 1 b))`, then the pattern synthesized by compilation of the lambda expression permits `(cons 1 b)` to be strict only in its first argument.

## Section 2.2: A lattice of strictness patterns

Strictness patterns can be defined as elements of a complete lattice that contains both finitely representable and infinite limit points. The set of useful strictness patterns is reduced to those patterns that can be represented by a finite graph, with or without cycles (cf. rationals, as opposed to irrational numbers).

All functions in the source and target language are regarded as taking one argument. This argument is similar to a Lisp S-expression (see Section 2.3.3). If the argument is a list, then different fields within it may be regarded as the function's arguments and the entire structure is then called the *argument collection.* For this reason, the definition of a strict function is expanded to include binary trees and specify an *index* for each part of the argument collection in which the function is strict. Consider the conventional labeling of a binary tree with root labeled '1', right children sucessively labeled with '1' and left children labeled with '0'. The index of each node in this tree is the number represented by the

concatenation of bit labels along the path from the root to its location. The following displays the indexing of a tree:

$$1\langle\ 2\langle 4..., 5...\rangle,\ 3\langle 6..., 7...\rangle\rangle\rangle.$$

In the following definition, $(n\ ac)$ selects an argument at index $n$ in the argument collection $ac$.

**Definition:** A function $f$ is strict in an argument $a$ at index $n$ of its argument collection $ac$ if $(n\ ac) = \bot \Longrightarrow f\ ac = \bot$.

A list marked with $ at any indexed sublist is to be evaluated by a suitably modified lazy interpreter using call-by-value for the marked field. Strictness at a given index does not necessarily imply that a function is strict at any other index of its argument. For example, the evaluation of

```
(cons $(cons b c) d)
```

is lazy in the the values of `b,c,d` but strict in the external structure of the pair `<b c>` (cf. Landin's stream construct [24]).

In other words, an expression is strict in a given sub-expression when the sub-expression and all containing structures are marked. For example, the program

```
(cons (cons $(cons a b) nil) nil)
```

is not strict in `a`, but

```
(cons $(cons $(cons $a b) nil) nil)
```

is strict in `a`.

### 2.2.1: Definition of P

Let $\$\mathbf{P} = \{\$\pi | \pi \in \mathbf{P}\}$, let $+$ connote coalesced sum, and let us require all lifting to be explicit [34]. Then the domain $\mathbf{P}$ may be defined by the reflexive equation,

$$\pi : \mathbf{P} = \$\mathbf{P} + (\mathbf{P} \times \mathbf{P})_\perp$$

**subject to** the homomorphic collapse required by the following two rules:

RULE 1. $\pi \sqsubseteq \$\pi$.

Strictness strengthens a pattern.

RULE 2. $\$\$\pi \sqsubseteq \$\pi$.

PROPOSITION. $\$\pi = \$\$\pi$.

Strictness is idempotent.

$\mathbf{P}$ is a complete lattice [35], with a top element,

$$fix\, \lambda\pi.\$\langle \pi\ ,\ \pi \rangle\ =\ \top_{\mathbf{P}}$$

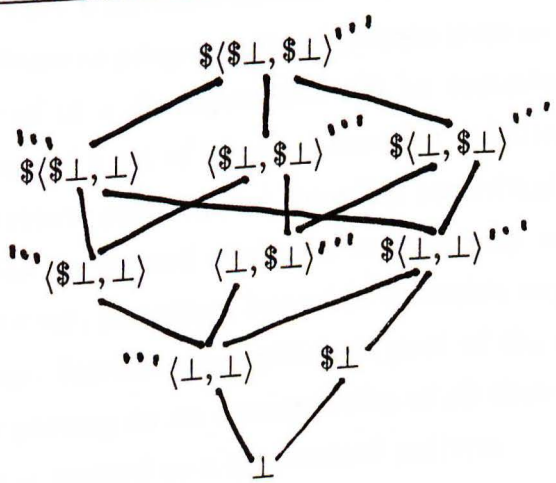
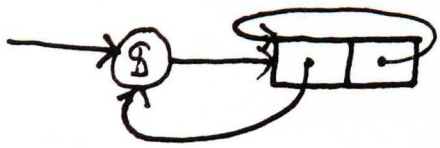



Figure 1. Partial lattice $\mathbf{P}$ of strictness patterns

An important element of **P** is the *printer* pattern

$$\pi_0 = \$fix\,\lambda\pi.\langle\$\pi\ ,\ \pi\rangle \neq \top_{\mathbf{P}}.$$

which can be abstractly represented as the finite cyclic graph:



The meet, join, and equality of two such patterns, represented as finite cyclic graphs, can be finitely computed (derived similarly to taking the intersection of regular expressions.) In the equations that follow, all patterns belong to the set of finitely representable elements in **P**, which form an (incomplete) sublattice.

### Section 2.3: The compiler

The compilation of a program inherits the *printer* pattern, $\pi_0$, which is strict only in the outer structure and the heads of all trees and sub-trees. This strictness pattern implies a leftmost-outermost evaluation order, and allows the compiler to find strictness in programs that generate trees as well as flat streams.

A function defined in a *fix* expression will be compiled into one or more versions — initially one of a set of identical function definitions that is uniquely associated with a strictness pattern, and compiled accordingly. Identifiers bound in a *fix* expression will be treated similarly, as they may represent recursively defined streams. However, identifiers bound in a *lambda* expression will not be converted into versions. Instead, they will form part of the mechanism for synthesizing patterns by passing on an accumulation of all their inherited patterns, which when complete is treated as a synthesized pattern.

The *join* of cyclic synthesized patterns might produce a pattern higher in the sublattice of **P** than $\pi_0$ (see ($\mathcal{C}$ 12) below), so any synthesized pattern is created by taking the *meet* of any least upper bound with the printer pattern itself. Some restriction of this kind is needed to avoid marking recursive calls that create the tails of infinite lists; however it is also possible to propagate a

similar initial pattern with a cycle containing several marked tails, as long as one remains unmarked in the cycle. $\pi_0$ will be used here for the sake of simplicity.

The *meet* of synthesized patterns with $\pi_0$ guarantees that $\pi_0$ is an upper bound for all patterns synthesized during compilation. Aside from the printer pattern, synthesized patterns are the only patterns that need to be controlled in this way, because only they represent potentially new cyclic patterns that may cause recursive versions to be created as the patterns are unrolled during compilation.

One consequence of restricting patterns in this way is that some list constructors may not be marked as fully as possible, and that strictness marks may appear inside sublists but not on the surrounding list expression. The effect produced by this is that the interior marks will affect the efficiency of the list evaluation when the *sublist* is evaluated. Except for conditional expressions, which contain marks that may not affect executed code because both branches are compiled while only one is executed at runtime, interior marks indicate that the the surrounding sub-expression will in fact eventually require the value of the marked expression.

The compiler builds up strictness information about identifiers and functions in a compile-time environment. It receives an integer resource that bounds the number of different versions that can be created for any given function or identifier. This limit is essential, even when versions inherit only rational patterns, because it is still possible to construct a pathological function that, when compiled, causes an infinite number of versions to be created, all of which are associated with unique rational patterns.

Function invocations that can be recognized as references to extant versions consume no additional resources.

The abstract compiler presented here marks only strict **cons** arguments, as Daisy is a strict interpreter.

### 2.3.1: Restricted Daisy syntactic categories

| | |
|---|---|
| | (identifiers) |
| id $\in$ *IDE* ; | (syntactic expressions) |
| e $\in$ *EXP* ; | (constants) |
| const $\in$ CONST. | |

### 2.3.2: Restricted Daisy syntax

| | |
|---|---|
| e ::= expr \| $expr | *constants* |
| expr ::= const \| | *nil* |
| []\| | *lists* |
| ⟨ exprs ⟩\| | *primitives with 2 arguments* |
| prim:⟨e e⟩ \| | *head application* |
| head:e \| | *tail application* |
| tail:e \| | *conditional application* |
| if:⟨e e e⟩ \| | *lambda application* |
| (λ id. e) : e \| | *application of a recursive function* |
| (fix :[id λ id. e]):e \| | *data recursion* |
| fix :[id e] \| | *function application* |
| id:e \| | *identifiers* |
| id \| | *infinite loop* |
| bottom | |

exprs ::= e exprs | e . e | *empty*

Expressions surrounded by double brackets are syntactic expressions in the source and target language. Syntactic expressions will often contain numbered sub-expressions, so that they are easier to discuss. For example, ⟦`prim:<e e>`⟧ becomes ⟦`prim:<e1 e2>`⟧.

### 2.3.3: Restricted Daisy value domains and semantic functions

| | |
|---|---|
| | (atoms) |
| A; | (structures) |
| $S = A + (S \times S) + (S \longrightarrow S)$. | |

Johnson presents a denotational semantics for Daisy [19]. However, denotational semantics can say nothing about the order of evaluation—early or delayed—upon which this work focuses. If the reader were willing to absorb a formal operational semantics for Daisy, it might be possible to formally argue the relative performance of the compiler's source and object code. No such semantics is presented here.

Colon is an infix *apply* operator occuring between *function* and *argument*; angle brackets construct lists, like Lisp's `list`. A function argument is either an atom or a list. `fix` expressions can be alternatively read with an (understood) outermost $\lambda$ and infix period wrapping the bracketed structure.

### 2.3.4: Compiler domains

$$C : \quad D \longrightarrow D; \qquad \text{(compiler)}$$

$$D = EXP \times \mathbf{P} \times ENV \times INT; \qquad \text{(compilation data)}$$

$$\pi : \quad \mathbf{P} = \$\mathbf{P} + (\mathbf{P} \times \mathbf{P}); \qquad \text{(strictness patterns)}$$

$$\rho : \quad ENV = V \longrightarrow (BEXP \times PF \times INT \times BTAG) + unbound; \qquad \text{(compiler environment)}$$

$$\iota : \quad INT; \qquad \text{(resource)}$$

$$\nu : \quad V = ID + (ID \times \mathbf{P}); \qquad \text{(version identifiers)}$$

$$BEXP = [\![[]]\!] + [\![(\text{fix} : [\text{id1} \quad \lambda \text{ id2. e1}]) : \text{e2}]\!] + [\![\text{fix} : [\text{id e}]]\!]$$

$$pa : \quad PF = (\mathbf{P} \longrightarrow (\mathbf{P}_{\pi_0} + unbound)) + \mathbf{P}_{\pi_0}; \qquad \text{(inherited and synthesized pattern entries in environment)}$$

$$\mathbf{P}_{\pi_0} = \{\pi \in \mathbf{P} \mid \pi \sqsubseteq \pi_0\};$$

$$BTAG = \mathbf{lambda} + \mathbf{fix}$$

(binding tags in environment)

The following functions are projections on environment entries:

$Binding = \lambda\, e.\ e{\downarrow}1$

$Pat\text{-}fun = \lambda\, e.\ e{\downarrow}2{\downarrow}1$

$Binding\text{-}type = \lambda\, e.\ e{\downarrow}2{\downarrow}2{\downarrow}2{\downarrow}1$

The compiler is given a syntactic expression, a strictness pattern, a compiler environment that performs some bookkeeping, and a natural number that limits the number of versions to be created for any one function. The domain of strictness patterns, of which $\mathbf{P}_{\pi_0}$ is a subset, has already been defined, but the domain of environments has interesting structure which is described in more detail in the following section.

### 2.3.5: Domain of compiler environments

The compiler environment allows the compiler to predict the scope in which expressions will appear at run-time. An entry for a given identifier contains a syntactic expression (either a particular *fix* expression encountered during compilation or a dummy expression $[\![\,]\!]$), inherited and synthesized pattern information which may either be a pattern or a function from patterns to patterns, a version count, and a tag indicating that the identifier was bound in either a lambda expression (**lambda**) or a recursively defined *fix* expression (**fix** ).

The domain of environments, *ENV* , contains only environments with pattern entries that are at most $\pi_0$ if simple patterns, and that, if functions, map elements from $\mathbf{P}$ into $\mathbf{P}_{\pi_0}$, a sub-lattice of $\mathbf{P}$ whose top element is $\pi_0$.

## Section 2.4: Compiler semantic functions

These equations describe an operational semantics for the abstract compiler. Examples appear after some of the equations. These examples present the code

(the other parts of the tuple are omitted) produced by the compiler when it receives an expression, inherited pattern, environment and resource. The environment is assumed to be the initial environment, but the expression, inherited pattern and resource will be described. The pattern

$$\$fix\,\lambda\pi.\langle\$\pi\ ,\ \pi\rangle$$

is the pattern to be initially propagated by the compiler, however the examples in both this and the next section use a variety of patterns. (Note that this pattern contains a mark *outside* the scope of the *fix* expression defining it; this is not the same pattern as

$$fix\lambda\pi.\$\langle\$\pi,\pi\rangle,$$

which represents a pattern with marked tails.)

### 2.4.1: Notation

The following notation is introduced:

— $\alpha{\cdot}\pi$ represents a strictness pattern, $\pi$, that may or may not be prefixed with \$. If \$ is the prefix, then $\alpha = \$$, otherwise $\alpha$ is the null string.

— A single vertical bar indicates concatenation of an identifier with a strictness pattern; this forms a new name defined locally within each rule.

— Unsubscripted $\perp$ stands for $\perp_{\mathrm{P}}$.

— ($\$ \in \alpha{\cdot}\pi$) denotes a strictness pattern in which a strictness mark is concatenated to some structure or sub-structure.

— The insertion of strictness marks during the compilation process is idempotent, and expressions marked by the programmer may be compiled.

— $[\![\mathtt{e}]\!]\,[\,[\![\mathtt{a}]\!]\,/\,[\![\mathtt{e}']\!]\,]$ refers to the substitution of $[\![\mathtt{a}]\!]$ for all instances of $[\![\mathtt{e}']\!]$ in $[\![\mathtt{e}]\!]$.

The compilation of the distinguished expression `bottom` is as follows:

$$C\,[\![\mathtt{bottom}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota = [\![\mathtt{bottom}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota$$

Note that the compilation of $[\![\mathtt{bottom}]\!]$ contributes no new pattern to any entry in the compiler environment.

### 2.4.2: Compiler rules

$$C\ [\![\texttt{const}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota = [\![\texttt{\$const}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota. \qquad (\mathcal{C}\ 1)$$

Constants cannot cause an infinite loop, so they can always be safely marked.

$$C\ [\![\texttt{e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota, \textbf{where}\ (\$ \notin \alpha{\cdot}\pi) = \qquad (\mathcal{C}\ 2)$$

$$\begin{cases} [\![\texttt{e}[\,[\![\texttt{fix:[id exp]}]\!] / [\![\texttt{e}']\!]\,]]\!]\ \alpha{\cdot}\pi\ \rho\ \iota \\ \textbf{where}\ [\![\texttt{fix:[id exp]}]\!] = \\ \quad (\textit{Binding}\ \ (\rho\ [\![\texttt{e}']\!])) \\ \qquad\qquad \text{if } \exists\ [\![\texttt{e}']\!] \in [\![\texttt{e}]\!]\ \textit{such that}\ [\![\texttt{e}']\!] \in \textit{ID} \\ \qquad\qquad \&\ (\textit{Binding-type}\ (\rho\ [\![\texttt{e}']\!])) = \textbf{fix}\ , \\ \\ [\![\texttt{e}[\,[\![\texttt{(fix:[f }\lambda\texttt{id. body])}\texttt{:exp}]\!] / [\![\texttt{e}']\!]\,]]\!]\ \alpha{\cdot}\pi\ \rho\ \iota \\ \textbf{where}\ [\![\texttt{(fix:[f }\lambda\texttt{id. body])}]\!] = \\ \quad (\textit{Binding}\ \ (\rho\ [\![\texttt{f}]\!])) \\ \qquad\qquad \text{if } \exists\ [\![\texttt{e}']\!] \in [\![\texttt{e}]\!]\ \textit{such that}\ [\![\texttt{e}']\!] = [\![\texttt{f:exp}]\!] \\ \qquad\qquad \&\ (\textit{Binding-type}\ (\rho\ [\![\texttt{f}]\!])) = \textbf{fix}\ , \\ \\ [\![\texttt{e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota \\ \qquad\qquad \text{otherwise.} \end{cases}$$

This rule is required in order to ensure that lazy references to renamed fix expressions do not refer to names which no longer exist— it is necessary for bookkeeping purposes. Essentially, compilation stops once the propagated strictness pattern cannot improve the source code.

$$C\ [\![\texttt{head:e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota = [\![\texttt{head:e}_1]\!]\ \alpha{\cdot}\pi\ \rho_1\ \iota \qquad (\mathcal{C}\ 3)$$
$$\textbf{where}\ [\![\texttt{e}_1]\!]\ \alpha_1{\cdot}\pi_1\ \rho_1\ \iota_1 = C\ [\![\texttt{e}]\!]\ \alpha{\cdot}\langle\alpha{\cdot}\pi,\ \bot\rangle\ \rho\ \iota.$$

$$C\,[\![\texttt{tail:e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota = [\![\texttt{tail:e}_1]\!]\ \alpha{\cdot}\pi\ \rho_1\ \iota \qquad (\mathcal{C}\ 4)$$
$$\textbf{where}\ [\![\texttt{e}_1]\!]\ \alpha_1{\cdot}\pi_1\ \rho_1\ \iota_1 = C\,[\![\texttt{e}]\!]\ \alpha{\cdot}\langle\perp,\ \alpha{\cdot}\pi\rangle\ \rho\ \iota.$$

Patterns inherited by applications of [[head]] or [[tail]] are injected into a list pattern to eventually be inherited by a list. For example, if $exp =$

[[head:<head:<a . b> . tail:<c . d>>]]

then $C\,[\![exp]\!]$ \$*fix* $\lambda\pi.\langle\$\pi\ ,\ \pi\rangle\ \rho\ 4 =$

[[head:<$head:<$a . b> . tail:<c . d>>]].

If $exp =$

[[tail:<head:<a . b> . tail:<c . d>>]]

then $C\,[\![exp]\!]$ \$*fix* $\lambda\pi.\langle\$\pi\ ,\ \pi\rangle\ \rho\ 4 =$

[[tail:<head:<a . b> . $tail:<c . $d>>]].

$$C\,[\![\texttt{<e1 . e2>}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota = \qquad (\mathcal{C}\ 5)$$
$$\begin{cases} C\,[\![\texttt{<e1 . e2>}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota, \textbf{where}\ (\$ \notin \alpha{\cdot}\pi) & \text{if } (\$ \notin \pi); \\ [\![\texttt{<}\alpha_1{\cdot}\texttt{e1}_1\ .\ \alpha_2{\cdot}\texttt{e2}_2\texttt{>}]\!]\ \alpha{\cdot}\pi\ \rho_2\ \iota, & \text{otherwise;} \end{cases}$$
$$\textbf{where}$$
$$\alpha_1{\cdot}\pi_1 = (\pi{\downarrow}1)$$
$$\alpha_2{\cdot}\pi_2 = (\pi{\downarrow}2)$$
$$[\![\texttt{e1}_1]\!]\ \alpha_1{\cdot}\pi_1\ \rho_1\ \iota_1 = C\,[\![\texttt{e1}]\!]\,(\pi{\downarrow}1)\ \rho\ \iota;$$
$$[\![\texttt{e2}_2]\!]\ \alpha_2{\cdot}\pi_2\ \rho_2\ \iota_2 = C\,[\![\texttt{e2}]\!]\,(\pi{\downarrow}2)\ \rho_1\ \iota.$$

The compilation of **cons** passes the head of its inherited pattern to the compilation of its first argument and then the tail of the inherited pattern to the compilation of its second argument. Preorder traversal is implied by an inherited pattern under $\pi_0$. (Note that when $\pi$ doesn't contain a strictness mark, the second compiler rule is executed.)

For example, if $exp =$

[[<head:<a . b> . tail:<c . d>>]]

then $C\,[\![exp]\!]$ \$*fix* $\lambda\pi.\langle\$\pi\ ,\ \pi\rangle\ \rho\ 4 =$

$[\![$<$head:<$a . b> . tail:<c . d>>$]\!]$.

$$C\,[\![\texttt{prim:<e1 e2>}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota = [\![\texttt{prim:}\mathtt{e}_1]\!]\ \alpha{\cdot}\pi\ \rho_1\ \iota \qquad (\mathcal{C}\,6)$$

**where**

$$[\![\mathtt{e}_1]\!]\ \alpha_1{\cdot}\pi_1\ \rho_1\ \iota_1 = C\,[\![\texttt{<e1 e2>}]\!]\ \langle \$\bot\ , \$\langle\ \$\bot, \bot\rangle\rangle\ \rho\ \iota.$$

Primitives (arithmetic and logical) are strict in both arguments.

For example, if $exp =$

$[\![$add:<head:<a . b> tail:<c . d>>$]\!]$

then $C\,[\![exp]\!]\ \$\bot\ \rho\ 4 =$

$[\![$add:<$head:<$a . b> $tail:<c . $d>>$]\!]$.

Note that a slightly different strictness notation for lists is introduced here. $[\![$<$x $y>$]\!]$ is strict in both $[\![$x$]\!]$ and $[\![$y$]\!]$ — the meaning is the same as $[\![$<$x . $<$y . $<>>>$]\!]$.

$$C\,[\![\texttt{if:<e1 e2 e3>}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota \qquad (\mathcal{C}\,7)$$
$$= [\![\texttt{if:<}\$\mathtt{e1}_1\ \mathtt{e2}_2\ \mathtt{e3}_3\texttt{>}]\!]\ \alpha{\cdot}\pi\ \rho_4\ \iota$$

**where**

$[\![\mathtt{e1}_1]\!]\ \alpha_1{\cdot}\pi_1\ \rho_1\ \iota_1 = C\,[\![\texttt{e1}]\!]\ \$\bot\ \rho\ \iota$;
$[\![\mathtt{e2}_2]\!]\ \alpha_2{\cdot}\pi_2\ \rho_2\ \iota_2 = C\,[\![\texttt{e2}]\!]\ \alpha{\cdot}\pi\ \rho_1\ \iota$;
$[\![\mathtt{e3}_3]\!]\ \alpha_3{\cdot}\pi_3\ \rho_3\ \iota_3 = C\,[\![\texttt{e3}]\!]\ \alpha{\cdot}\pi\ \rho_1\ \iota$;

$$\rho_4 = \lambda i. \begin{cases} \langle\, [\![\mathtt{binding}_2]\!]\,,\ (pa_2 \sqcap pa_3)\ , 0, \textit{b-type}_2\rangle \\ \qquad \text{if } \textit{b-type}_2 = \textbf{lambda}; \\ \langle\, [\![\mathtt{binding}_2]\!]\,, pa_4, \textit{v-count}_2 + \textit{v-count}_3, \textit{b-type}_2\rangle \\ \qquad \text{if } \textit{b-type}_2 = \textbf{fix}\ ; \end{cases}$$

**where**

$\langle\, [\![\mathtt{binding}_2]\!]\,, pa_2, \textit{v-count}_2, \textit{b-type}_2\rangle = \rho_2\, i$
$\langle\, [\![\mathtt{binding}_3]\!]\,, pa_3, \textit{v-count}_3, \textit{b-type}_3\rangle = \rho_3\, i$

$$pa_4 = \lambda pat. \begin{cases} (pa_3\ pat) & \text{if } (pa_2\ pat) = \textit{unbound}; \\ (pa_2\ pat) & \text{otherwise.} \end{cases}$$

A conditional expression is strict in its predicate, but not in any of the paths of the predicate's result. Each branch of the **if** may safely be compiled using the pattern inherited by the **if** application as long as the leading $ is stripped off the compiled code when the new application is assembled and returned. This permits the predicate to be evaluated before either of the two branches, and allows the selected branch to be as efficient as possible. The new environment returns the *meet* of the patterns inherited from either branch by a variable bound in a lambda expression, or the appropriate entry. (The compilation of conditional expressions receives special attention in Chapter 4, where a more powerful equation is developed and presented.)

For example, if $exp =$

```
[[if:<same?:<head:<a . b> tail:<c . d>>
      <head:<d . e> . d>
      <b . c>]]
```

then $C\ [\![exp]\!]\ \langle \$\bot\ ,\ \$\bot \rangle\ \rho\ 1 =$

```
[[if:<$same?:<$head:<$a . b> $tail:<c . $d>>
      <$head:<$d . e> . $d>
      <$b . $c>>]].
```

$$
\begin{aligned}
&C\ [\![(\lambda \texttt{id. body}) \texttt{:e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota = \qquad (C\ 8)\\
&\qquad [\![(\lambda \texttt{id. body}_1) \texttt{:e}_1]\!]\ \alpha{\cdot}\pi\ \rho_4\ \iota\\
&\textbf{where}\\
&[\![\texttt{body}_1]\!]\ \alpha_1{\cdot}\pi_1\ \rho_1\ \iota_1 = C\ [\![\texttt{body}]\!]\ \alpha{\cdot}\pi\ \rho_2\ \iota\\
&\rho_2 = \lambda i. \begin{cases} \langle\ [\![\texttt{[]}\ ]\!], \bot, 0, \textbf{lambda}\rangle, & \text{if } i = [\![\texttt{id}]\!]\,;\\ \rho\, i, & \text{otherwise;} \end{cases}\\
&\rho_3 = \lambda i. \begin{cases} \rho\, i, & \text{if } i = [\![\texttt{id}]\!]\,;\\ \rho_1\, i, & \text{otherwise;} \end{cases}\\
&[\![\texttt{e}_1]\!]\ \alpha_2{\cdot}\pi_2\ \rho_4\ \iota_2 = C\ [\![\texttt{e}]\!]\ (\mathit{Pat\text{-}fun}\ (\rho_1\ [\![\texttt{id}]\!]))\ \rho_3\ \iota.
\end{aligned}
$$

Lambda applications demonstrate the need for the compiler environment, $\rho$. Initially, an entry is created for the identifier $[\![\texttt{id}]\!]$, which includes a (meaningless) syntactic expression, an initial inherited pattern, a version count (again meaningless), and a tag which indicates that $[\![\texttt{id}]\!]$ was bound in a **lambda** environment. As the compiler explores the body of the lambda expression, the pattern inherited by $[\![\texttt{id}]\!]$ is updated. When analysis of the body is complete, the identifier has inherited a composite pattern which becomes the synthesized pattern for this lambda expression. The projection funtion, *Pat-fun*, retrieves this pattern so that it can be propagated to the analysis of $[\![\texttt{e}]\!]$.

For example, if $exp =$

$[\![(\lambda\texttt{a. <head:a . head:a>):<b . c>}]\!]$

then $C\,[\![exp]\!]\ \langle \$\bot\ ,\ \$\bot\rangle\ \rho\ 1 =$

$[\![(\lambda\texttt{a. <\$head:a . \$head:a>):\ \ <\$b . c>}]\!]$

$$
\begin{aligned}
&C\,[\![\texttt{(fix:[f }\lambda\texttt{id. body])}\texttt{:e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota = \qquad (\mathcal{C}\,9)\\
&\qquad [\![\texttt{(fix:[f}|\alpha{\cdot}\pi\ \lambda\texttt{id. body}_1\texttt{])}\texttt{:e}_1]\!]\ \alpha{\cdot}\pi\ \rho_4\ \iota\\
&\textbf{where}\\
&[\![\texttt{body}_1]\!]\ \alpha_1{\cdot}\pi_1\ \rho_1\ \iota_1 = C\,[\![\texttt{body}]\!]\ \alpha{\cdot}\pi\ \rho_2\ \iota\\
&\rho_2 = \lambda i.\begin{cases}\langle\,[\![\texttt{(fix:[f }\lambda\texttt{id. body])}]\!]\,, pa, 1, \textbf{fix}\,\rangle, & \text{if } i = [\![\texttt{f}]\!]\,;\\ \langle\,[\![\texttt{[]}]\!]\,, \bot, 0, \textbf{lambda}\rangle, & \text{if } i = [\![\texttt{id}]\!]\,;\\ \rho\, i, & \text{otherwise;}\end{cases}\\
&pa = \lambda pat.\begin{cases}\textit{rec-p} & \text{if } pat = \alpha{\cdot}\pi;\\ \textit{unbound} & \text{otherwise;}\end{cases}\\
&\rho_3 = \lambda i.\begin{cases}\rho\, i, & \text{if } i = [\![\texttt{id}]\!] \textit{ or } i = [\![\texttt{f}]\!]\,;\\ \rho_1\, i, & \text{otherwise;}\end{cases}\\
&\textit{rec-p} \;=\; [\![\texttt{e}]\!] \bullet C\,[\![(\sqcup_{u=1}^{\infty}(U_u\ \texttt{(fix:[f }\lambda\texttt{id. body])}))\texttt{:e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota\\
&\qquad = (\rho_1\ [\![\texttt{id}]\!])\!\downarrow\!2\!\downarrow\!1\\
&[\![\texttt{e}_1]\!]\ \alpha_2{\cdot}\pi_2\ \rho_4\ \iota_2 = C\,[\![\texttt{e}]\!]\ \textit{rec-p}\ \rho_3\ \iota.
\end{aligned}
$$

The compiler constructs a synthesized pattern *rec-p* by recursively defining the result of the analysis of the $\lambda$ body. This pattern is then inherited by the argument $[\![\texttt{e}]\!]$. $U$ is defined and discussed in Chapter 3. (Section 2.6 on the implementation of the compiler discusses the significant problem that arises when pattern bindings are not maintained as explicitly labelled objects by the compiler.)

For example, if $exp =$

```
[[(fix:[f
  λlst.
    <add:<head:lst head:tail:lst>
     . f:tail:lst>]):a]]
```

then $C\,[\![exp]\!]\ fix\,\lambda\pi.\langle\$\bot\ ,\ \pi\rangle\ \lambda id.unbound\ 1 =$

```
[[(fix:[f-p1
  λlst.
    <$add:<$head:lst $head:tail:lst>
     . f-p1:tail:lst>]):a]]
```

where `p1`$= fix\,\lambda\pi.\langle\$\bot\ ,\ \pi\rangle$.

$[\![\texttt{a}]\!]$ then inherits the pattern $\$fix\,\lambda\pi.\langle\$\bot\ ,\ \pi\rangle$.

$$C\,[\![\texttt{fix:[id e]}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota = [\![\texttt{fix:[id|}\alpha{\cdot}\pi\ \texttt{e}_1\texttt{]}]\!]\ \alpha{\cdot}\pi\ \rho_3\ \iota_1 \qquad (\mathcal{C}\ 10)$$

**where**

$$[\![\texttt{e}_1]\!]\ \alpha_1{\cdot}\pi_1\ \rho_1\ \iota_1 = C\,[\![\texttt{e}]\!]\ \alpha{\cdot}\pi\ \rho_2\ \iota;$$

$$\rho_2 = \lambda i.\begin{cases}\langle\,[\![\texttt{fix:[id e]}]\!], pa, 1, \textbf{fix}\,\rangle, \\ \qquad \text{if } i = [\![\texttt{id}]\!]; \\ \rho\, i, \text{otherwise};\end{cases}$$

$$pa = \lambda pat.\begin{cases}\bot & \text{if } pat = \alpha{\cdot}\pi; \\ unbound & \text{otherwise};\end{cases}$$

$$\rho_3 = \lambda i.\begin{cases}\rho\, i, & \text{if } i = [\![\texttt{id}]\!]; \\ \rho_1\, i, & \text{otherwise.}\end{cases}$$

This equation permits the construction of recursively defined lists.

For example, if $exp =$

$[\![$`fix:[1 <a . 1>]`$]\!]$

then $C\,[\![exp]\!]\,\langle \$\bot\ ,\ \langle \bot\ ,\ fix\,\lambda\pi.\langle \$\bot\ ,\ \pi\rangle\rangle\rangle\ \lambda id.unbound\ 1 =$

$[\![$`fix:[1-p1`
`<$a .`
`fix:[1-p2`
`<a . fix:[1 <a . 1>]>]>]`$]\!]$

and $C\,[\![exp]\!]\ \langle \$\bot\ ,\ \langle \bot\ ,\ fix\,\lambda\pi.\langle \$\bot\ ,\ \pi\rangle\rangle\rangle\ \lambda id.unbound\ 2 =$

$[\![$`fix:[1-p1`
`<$a .`
`fix:[1-p2`
`<a . fix:[1-p3 <$a . 1-p3>]>]>]`$]\!]$

where

p1= $\langle \$\bot\ ,\ \langle \bot\ ,\ fix\,\lambda\pi.\langle \$\bot\ ,\ \pi\rangle\rangle\rangle$;

p2= $\langle \bot\ ,\ fix\,\lambda\pi.\langle \$\bot\ ,\ \pi\rangle\rangle$;

p3= $fix\,\lambda\pi.\langle \$\bot\ ,\ \pi\rangle$.

In the first example, the compiler can make only the first two elements of the output stream strict because the number of versions is too small to permit it to discover the recursive loop. When it is allowed one more version, it is able to make the entire stream strict in its heads. A resource of three or more would still produce the result from the second example.

$(\mathcal{C}\ 11)$

$$C\ [\![\texttt{f:e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota =$$

$$\begin{cases} \textbf{Reached-Limit} & \text{if } (pa\ \alpha{\cdot}\pi) = \textit{unbound}\ \&\ \textit{v-count} \geq \iota; \\ \textbf{Compile-Binding} & \text{if } (pa\ \alpha{\cdot}\pi) = \textit{unbound}\ \&\ \textit{v-count} < \iota; \\ \textbf{Mark-With-Pattern} & \text{otherwise;} \end{cases}$$

$$\textbf{where} \langle\ [\![\texttt{(fix:[f }\lambda\texttt{id. body])}]\!], pa, \textit{v-count}, \textbf{fix}\ \rangle = \rho\ [\![\texttt{f}]\!]\ ;$$

**Reached-Limit is** $[\![\texttt{(fix:[f }\lambda\texttt{id.body]):e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota$;

**Compile-Binding is**
$[\![\texttt{(fix:[f}|\alpha{\cdot}\pi\ \lambda\texttt{id. body}_1\texttt{]):e}_1]\!]\ \alpha{\cdot}\pi\ \rho_4\ \iota$
**where**

$$[\![\texttt{body}_1]\!]\ \alpha_1{\cdot}\pi_1\ \rho_1\ \iota_1 = C\ [\![\texttt{body}]\!]\ \alpha{\cdot}\pi\ \rho_2\ \iota$$

$$\rho_2 = \lambda i. \begin{cases} \langle\ [\![\texttt{(fix:[f }\lambda\texttt{id. body])}]\!], pa_1, \textit{v-count}+1, \textbf{fix}\ \rangle, & \text{if } i = [\![\texttt{f}]\!]\ ; \\ \langle\ [\![\texttt{[]}]\!], \bot, 0, \textbf{lambda}\rangle, & \text{if } i = [\![\texttt{id}]\!]\ ; \\ \rho\ i, & \text{otherwise;} \end{cases}$$

$$pa_1 = \lambda pat. \begin{cases} \textit{rec-p} & \text{if } pat = \alpha{\cdot}\pi; \\ pa & \text{otherwise;} \end{cases}$$

$$\rho_3 = \lambda i. \begin{cases} \rho\ i, & \text{if } i = [\![\texttt{id}]\!] \textit{ or } i = [\![\texttt{f}]\!]\ ; \\ \rho_1\ i, & \text{otherwise;} \end{cases}$$

$$\textit{rec-p} = [\![\texttt{e}]\!] \bullet C\ [\![(\sqcup_{u=1}^{\infty}(U_u\ \texttt{(fix:[f }\lambda\texttt{id. body])}))\texttt{:e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota$$

$$= (\rho_1\ [\![\texttt{id}]\!]\ )\downarrow 2 \downarrow 1$$

$$[\![\texttt{e}_1]\!]\ \alpha_2{\cdot}\pi_2\ \rho_4\ \iota_2 = C\ [\![\texttt{e}]\!]\ \textit{rec-p}\ \rho_3\ \iota$$

**Mark-With-Pattern is**
$[\![\texttt{f}|\alpha{\cdot}\pi\texttt{:e}_1]\!]\ \alpha{\cdot}\pi\ \rho_1\ \iota$
**where**

$$[\![\texttt{e}_1]\!]\ \alpha_1{\cdot}\pi_1\ \rho_1\ \iota_1 = C\ [\![\texttt{e}]\!]\ (pa\ \alpha{\cdot}\pi)\ \rho\ \iota$$

There are three possible ways in which recursive function applications can be compiled.

— If the version count for this particular function has been exhausted and the combination of this function call and the strictness pattern currently inherited has not been seen before, then the compiler expands the expression

once, guaranteeing that the lazy call refers to the correct name, and stops exploring the source code.

— If the version count for this particular function has *not* been exhausted and the combination of this function call and the strictness pattern currently inherited has not been seen before, then a new version is compiled. The current compiler environment is updated so that the function mapping an inherited pattern to a synthesized pattern for each version created so far now has an entry for this new version.

— Otherwise, the compiler is currently compiling a version whose compilation originally inherited the pattern propagated to the current function application. In this case, the synthesized pattern for this function is already known and can be retrieved from the environment entry for this function to be used in compiling the argument $[\![e]\!]$. See the previous examples.

(C 12)

$C\ [\![\texttt{id}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota =$

$$\begin{cases} \textbf{Variable} & \text{if } b\text{-}type = \textbf{lambda}; \\ \textbf{Reached-Limit} & \text{if } (pa\ \alpha{\cdot}\pi) = unbound\ \&\ v\text{-}count \geq \iota; \\ \textbf{Compile-Binding} & \text{if } (pa\ \alpha{\cdot}\pi) = unbound\ \&\ v\text{-}count < \iota; \\ \textbf{Mark-With-Pattern} & \text{otherwise;} \end{cases}$$

**where** $\langle\ [\![\texttt{binding}]\!]\ , pa, v\text{-}count, b\text{-}type\rangle = \rho\ [\![\texttt{id}]\!]\ ;$

**Variable is**

$[\![\texttt{id}]\!]\ \alpha{\cdot}\pi\ \rho_1\ \iota$

**where**

$$\rho_1 = \lambda i. \begin{cases} \langle\ [\![\texttt{binding}]\!]\ ,\ (\alpha{\cdot}\pi \sqcup pa) \sqcap \pi_0, 0, b\text{-}type\rangle, \\ \qquad \text{if } i = \texttt{id}; \\ \rho\ i, \text{otherwise;} \end{cases}$$

**Reached-Limit is** $[\![\texttt{binding}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota;$

**Compile-Binding is**

$[\![\texttt{fix:[id}|\alpha{\cdot}\pi\ \ \texttt{e}_1\texttt{]}]\!]\ \alpha{\cdot}\pi\ \rho_3\ \iota$

**where**

$[\![\texttt{fix:[id e]}]\!] = [\![\texttt{binding}]\!]$

$[\![\texttt{e}_1]\!]\ \alpha_1{\cdot}\pi_1\ \rho_1\ \iota_1 = C\ [\![\texttt{e}]\!]\ \alpha{\cdot}\pi\ \rho_2\ \iota;$

$$\rho_2 = \lambda i. \begin{cases} \langle\ [\![\texttt{fix:[id e]}]\!]\ , pa_1, v\text{-}count + 1, \textbf{fix}\ \rangle, \\ \qquad \text{if } i = [\![\texttt{f}]\!]\ ; \\ \rho\ i, \text{otherwise;} \end{cases}$$

$$pa_1 = \lambda pat. \begin{cases} \bot & \text{if } pat = \alpha{\cdot}\pi; \\ unbound & \text{otherwise;} \end{cases}$$

$$\rho_3 = \lambda i. \begin{cases} \rho\ i, & \text{if } i = [\![\texttt{id}]\!]\ ; \\ \rho_1\ i, & \text{otherwise.} \end{cases}$$

**Mark-With-Pattern is** $[\![\texttt{id}|\alpha{\cdot}\pi]\!]\ \alpha{\cdot}\pi\ \rho\ \iota$

Identifiers may be recursively bound to a value, in which case they are treated like recursive data (but in cases which are similar to the discussion above) or they may be bound in a lambda expression, in which case the pattern currently being inherited is combined with the pattern accumulated by earlier compilation of instances of the identifier in a lambda body. See the previous examples.

## Section 2.5: Three examples

The following is a simple and common program, in which a filter passes on certain elements of its argument stream.

```
[[(fix:[G λ1.
     <head:1 . G:tail:tail:1>]):
   (fix:[F λx.
     if:<eq?:<head:x 1>
           <head:x . F:<tail:x . head:x>>
           <(fix:[Bad λy. Bad:y]):head:x
              . F:<tail:x . head:x>>>]):
 <1 . 0> ]]
```

[[F]] produces a stream of alternating 1's and $\perp_S$. [[G]] selects odd elements of [[F]]'s result, avoiding the divergent elements. The compiler produces the following compiled expression, given $fix\,\lambda\pi.\langle\$\pi\ ,\ \pi\rangle$, the initial environment $\lambda id.unbound$, and the resource 5;

```
[[(fix:[G-p0 λ1.
     <$head:1 . G-p0:tail:tail:1>]):
   (fix:[F-p1 λx.
     if:<$eq?:<$head:x $1>
           <$head:x . F-p2:<$tail:x . head:x>>
           <$(fix:[Bad-p4 λy. Bad-p4:y]):head:x
              . F-p2:<$tail:x . head:x>>>]):
 <$1 . $0> ]]
```

where [[F-p2]] =

```
[[(fix:[F-p2 λx.
  if:<$eq?:<$head:x $1>
        <head:x . F-p3:<$tail:x . head:x>>
        <(fix:[Bad λy. Bad:y]):head:x
           . F-p3:<$tail:x . head:x>>>])]]
```

and $[\![\texttt{F-p3}]\!] =$

```
[[(fix:[F-p3 λx.
  if:<$eq?:<$head:x $1>
        <$head:x . F-p2:<$tail:x . head:x>>
        <$(fix:[Bad-p4 λy. Bad-p4:y]):head:x
           . F-p2:<$tail:x . head:x>>>])]]
```

and where

$\texttt{p0} = \mathit{fix}\,\lambda\pi.\langle \$\pi\ ,\ \pi\rangle$

$\texttt{p1} = \$\mathit{fix}\lambda\pi.\langle \pi_0\ ,\ \langle \bot\ ,\ \pi\rangle\rangle$

$\texttt{p2} = \mathit{fix}\lambda\pi.\langle \bot\ ,\ \langle \pi_0\ ,\ \pi\rangle\rangle$

$\texttt{p3} = \mathit{fix}\lambda\pi.\langle \pi_0\ ,\ \langle \bot\ ,\ \pi\rangle\rangle$

$\texttt{p4} = \pi_0$

The versions of $[\![\texttt{F}]\!]$ produce a stream that is alternately strict and lazy in its heads, and $[\![\texttt{G}]\!]$ is strict in all elements it accesses, but produces a stream strict in all heads and lazy in the tails. Notice that three patterns, those patterns which distinguish between versions of $[\![\texttt{F}]\!]$, have

$$\mathit{fix}\lambda\pi.\langle \bot\ ,\ \langle \bot\ ,\ \pi\rangle\rangle$$

as their greatest lower bound. If no versions of $[\![\texttt{F}]\!]$ were produced here, it would not have been possible to find strictness in $[\![\texttt{F}]\!]$. Versions $[\![\texttt{F-p1}]\!]$ and $[\![\texttt{F-p3}]\!]$ are similar and could be coalesced into one version inheriting the *meet* of the two patterns. $[\![\texttt{Bad-p4}]\!]$ produces a synthesized pattern that is $\bot$. The effect of this

pattern would be obvious if ⟦Bad-p4⟧ was applied to an expression using list syntax, such as ⟦<a . b>⟧.

The following examples are hard to read using the *fix* notation defined so far, so *rec* will be used instead. One of the advantages of *rec* is that it is a recursive binding function that permits all versions to be gathered into the same scope, producing less object code. Formals are grouped together, as are actuals, rather than pairs of formals and actuals. One of the implemented compilers replaces *fix* with *rec*.

In addition, an **if** with more than two branches is introduced. Predicates following the first one may be marked, but since the mark is covered by the tail of the *if* argument, it won't be evaluated until the branch in which it appears is selected. Function formal arguments are now destructured into a flat list of bound variables, however the corresponding actuals are written as dotted pairs. The functions ⟦odd?⟧, ⟦number?⟧, ⟦identifier?⟧, ⟦Fn⟧, ⟦Args⟧, ⟦Body⟧, ⟦Formals⟧ and ⟦nil?⟧ are all assumed to produce the synthesized pattern $⊥. ⟦eq?⟧ and ⟦same?⟧ are treated as in (*C* 6).

The first is a function that prints the even Fibonacci numbers, seeded with two values from anywhere in the series.

```
[[λ[a b].
rec:[[h Addall Skip]
<
  h =
    <a . <b . Addall:<h . <tail:h>>>>
  Addall = λ[a b].
      <add:<head:a head:b> .
          Addall:<tail:a . <tail:b>>>
  Skip = λ[stream].
      if:<odd?:head:stream
            Skip:<tail:stream>
            <head:stream . Skip:<tail:stream>>>
>
in
Skip:<h>]]
```

When compiled with the pattern $fix\,\lambda\pi.\langle\$\bot\ ,\ \pi\rangle$, and a resource of 4, the compiler produces the following output;

```
⟦λ[a b].
rec:[[h-p2 h-p3 Addall-p1 Skip-p1]
<
  h-p2 =
    $<$a . <$b Addall-p1:<$h-p2 . <$tail:h-p3>>>>
  h-p3 =
     $<a . $<$b . Addall-p1:<$h-p2 . <$tail:h-p3>>>>
  Addall-p1 = λ[a b].
    <$add:<$head:a $head:b> .
          Addall-p1:<$tail:a . <$tail:b>>>
  Skip-p1 = λ[stream].
      if:<$odd?:head:stream
            Skip-p1:<$tail:stream>
            <$head:stream . Skip-p1:<$tail:stream>>>
>
in
 Skip-p1:<$h-p2>]⟧

p1 = fix λπ.⟨$⊥ , π⟩
p2 = $fix λπ.⟨$⊥ , π⟩
p3 = $⟨⊥ , $fix λπ.⟨$⊥ , π⟩⟩
```

⟦Skip-p1⟧ is strict in all the heads of its stream argument, and passes this pattern to the recursive data structure ⟦h-p2⟧. ⟦Addall-p1⟧ inherits the same pattern. Note that two versions of ⟦h⟧ are created because one inherits the cyclic pattern passed on from ⟦Addall-p1⟧ while the other's inherited pattern is lazy in its head but inherits the cyclic pattern in the tail.

The second function compiled using *rec* is a simple interpreter. Its language is restricted to constants, identifiers, **head**, **tail**, **cons**, **quote** (written as ∧), and functions of one argument.

```
⟦λ[input]. EVAL:<input . <[]>>⟧
```

where [[EVAL]] =

```
[[λ[exp env].
  if:<number?:exp
        exp
      identifier?:exp
        LOOKUP:<exp . <env>>
        APPLY:<Fn:exp .
            <EVAL:<Args:exp . <env>> . <env>>>
      >]]
```

and [[APPLY]] =

```
[[λ[fn args env].
  if:<nil?:args
        [error]
      same?:<^head fn>
        head:args
      same?:<^tail fn>
        tail:args
      same?:<^cons fn>
        <head:args . tail:args>
      identifier?:fn
        APPLY:<EVAL:<fn . <env>> . <args . <env>>>
        EVAL:<Body:fn .
            <MKENV:<Formals:fn . <args . <env>>>>>
      >]]
```

and $[\![$LOOKUP$]\!]$ =

```
[![λ[exp env].
  if:<same?:<exp head:head:env>
        tail:head:env
        LOOKUP:<exp . <tail:env>>
     >
```

and $[\![$MKENV$]\!]$ =

```
[![λ[formals actuals env].
  if:<nil?:actuals
        env
        <<head:formals . head:actuals> .
           MKENV:<tail:formals . <tail:actuals env>>>
     >]!]
```

When the interpreter is compiled with the pattern $\$\perp$, and the resource 3, the following output is produced;

```
[![λ[input]. EVAL-p1:<$input . <$[]>>]!]
  where
p1 = $⊥
```

where ⟦EVAL-p1⟧ =

```
⟦λ[exp env].                                                    . <
  if:<$number?:exp                                              . <
        exp                                                     . <
      $identifier?:exp                                          . <
        LOOKUP-p1:<$exp . <$env>>
        APPLY-p1:<$Fn:exp .
           <$EVAL-p1:<$Args:exp . <env>> . <env>>>
                                                          .  <>>>>>

      >⟧
```

and ⟦APPLY-p1⟧ =

```
⟦λ[fn args env].                                                . <
  if:<$nil?:args                                                . <
       [error]                                                  . <
     $same?:<$∧head $fn>                                        . <
       head:args                                                . <
     $same?:<$∧tail $fn>                                        . <
       tail:args                                                . <
     $same?:<$∧cons $fn>                                        . <
       <head:args . tail:args>                                  . <
     $identifier?:fn
       APPLY-p1:<$EVAL-p1:<$fn . <env>> .                       . <
            <$args . <env>>>
       EVAL-p1:<$Body:fn .
           <MKENV:<Formals:fn . <args .
                 <env>>>>>                          .  <>>>>>>>>>>>

     >⟧
```

and ⟦LOOKUP-p1⟧ =

```
⟦λ[exp env].
 if:<$same?:<$exp $head:head:env>                          . <
      tail:head:env                                        . <
      LOOKUP-p1:<$exp . <$tail:env>>                       . <>>>
   >⟧
```

and ⟦MKENV⟧ =

```
⟦λ[formals actuals env].
 if:<nil?:actuals                                          . <
      env                                                  . <
      <<head:formals . head:actuals> .
          MKENV:<tail:formals .
            <tail:actuals env>>>                           . <>>>
   >⟧
```

⟦EVAL-p1⟧ is strict only in its first argument, and ⟦APPLY-p1⟧ is strict only in its first two arguments. ⟦LOOKUP-p1⟧ is strict in all of its arguments. ⟦MKENV⟧ is never reached by the propagation of a strict pattern, so it remains unchanged.

### Section 2.6: Representation of $\perp_p$ and pattern fixed points

The equations shown here are not directly executable with conventional binding mechanisms. For example, in analyzing the expression

⟦(fix:[f λn. f:n]):3⟧

the compiler would find within equation (*C* 9) that *rec-p* is bound to "the value of *rec-p*" and loop indefinitely. Actually *rec-p* = $\perp_p$, but conventional binding machinery denies us the ability to detect this case. However, the compiler maintains a table of pattern bindings, permitting it to detect such a binding. Since only *rational patterns* arise here, every potential divergence manifests itself

in such a cycle. Therefore, any binding that would diverge because of indirect self-dependence must cycle through some binding in the table. It is the *second* visit to such an entry in the table (of bounded size) which determines that the value of *rec-p* is $\perp_{\mathrm{P}}$.

The fixed points of synthesized patterns are constructed using the following *fix* equation, which corresponds to ($\mathcal{C}$ 9) as follows:

$$
\begin{array}{l}
\mathcal{C}\,[\![\texttt{(fix:[f }\lambda\texttt{id. body])}\texttt{:e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota = \qquad (\mathcal{C}\ 9') \\
\quad [\![\texttt{(fix:[f}|\alpha{\cdot}\pi\ \lambda\texttt{id. body}_1\texttt{])}\texttt{:e}_1]\!]\ \alpha{\cdot}\pi\ \rho_4\ \iota \\
\textbf{where} \\
[\![\texttt{body}_1]\!]\ \alpha_1{\cdot}\pi_1\ \rho_1\ \iota_1 = \mathcal{C}\,[\![\texttt{body}]\!]\ \alpha{\cdot}\pi\ \rho_2\ \iota \\
\rho_2 = \lambda i.\begin{cases} \langle\, [\![\texttt{(fix:[f }\lambda\texttt{id. body])}]\!]\,, pa, 1, \textbf{fix}\,\rangle, & \text{if } i = [\![\texttt{f}]\!]\,; \\ \langle\, [\![\texttt{[]}]\!]\,, \perp, 0, \textbf{lambda}\rangle, & \\ & \text{if } i = [\![\texttt{id}]\!]\,; \\ \rho\, i, & \text{otherwise;} \end{cases} \\
pa = \lambda pat.\begin{cases} \textit{rec-p} & \text{if } pat = \alpha{\cdot}\pi; \\ \textit{unbound} & \text{otherwise;} \end{cases} \\
\rho_3 = \lambda i.\begin{cases} \rho\, i, & \text{if } i = [\![\texttt{id}]\!] \textit{ or } i = [\![\texttt{f}]\!]\,; \\ \rho_1\, i, & \text{otherwise;} \end{cases} \\
\textit{rec-p} = (\rho_1\ [\![\texttt{id}]\!]\,)\!\downarrow\!2\!\downarrow\!1 \\
[\![\texttt{e}_1]\!]\ \alpha_2{\cdot}\pi_2\ \rho_4\ \iota_2 = \mathcal{C}\,[\![\texttt{e}]\!]\ \textit{rec-p}\ \rho_3\ \iota.
\end{array}
$$

The fixed point *rec-p* is initially represented by a distinguished pattern, interpreted as $\perp_{\mathrm{P}}$ if its value is required before the entire *fix* expression has been compiled; this distinguished pattern is bound to the synthesized pattern in the table of strictness patterns when compilation of the lambda body is finished. Since the value of *rec-p* is often not known when a recursive function application is compiled, the compiler currently makes another pass to annotate the argument of this application. (Subsequent passes may further improve the compiled code, although the examples presented in this thesis have been compiled using only two passes.)

## Section 2.7: Compiler safety and termination

### 2.7.1: Weak and strong safety

Any implementation of a lazy list-processing language that improves performance only through an analysis of strictness sacrifices some semantic strength when printing a list containing $\perp_S$. The problem is that some element of a list may be $\perp_S$ and it might occur in a position that has been analyzed as "strict." Thus, an enveloping portion of the list may "diverge" (*i.e.* evaluate to $\perp_S$), even though a truly lazy implementation would have no difficulty with the structure of the envelope. In the simplest case, Landin's $\langle \$\perp_S \, . \, v \rangle$ (where $v$ is a value in S higher than $\perp_S$), this divergence causes the printing operation to lose even the outer left parenthesis, since it is assumed that the printer is as lazy as possible and will note the fact that the object being printed is a list before attempting to print the list's contents. For this reason, weak safety as defined in Chapter 1 is as much as can be expected in a compiler *unless* a special analysis is done in order to establish the order in which elements within a list will be printed, so that some troublesome cases can be avoided.

### 2.7.2: Stream output

A more complicated case, such as $F{:}\langle \$\perp_S \ \$1 \rangle$, where $F = \lambda a \ b. \ \langle \$b \, . \, \langle \$a \, . \, \langle\rangle\rangle\rangle$, would cause the loss of the output prefix '⟨1'.

This problem is inherent in streams, but streams create yet another problem. In some special cases, the printer will not print some prefix of the output even though none of the elements are $\perp_S$. For example, a stream of natural numbers can be created by the following expression:

```
[[F:0 where F = λn. <$n . F:inc:n>]]
```

It can also be created as follows:

```
[[F:0 where F = λn. <$F:inc:n . n>]]
```

which, of course, has no printable prefix. Traversal of the stream of naturals constructed the second way, with recursion in the left of the resulting list, will cause the loss of the initial parentheses produced by traversal over a lazy expression.

(This could be avoided if the printer pattern was weakened so that it was strict only in every other head, rather than strict in all heads, but an infinite series of parentheses appears meaningless enough to permit the use of the stronger pattern.) For these reasons, it seems best to define an *admissible* answer and show that for such results, the printer produces the same output as the traversal of a lazy expression.

From this point on, a statement that $C$ is safe means that $C$ is safe when its interpreted source code produces admissible values. If the user isn't interested in seeing the preceding elements of a list that contains $\perp_S$, or in seeing an infinite series of left parentheses, then $C$ produces useful results even when its interpreted source code is not admissible.

### 2.7.3: Admissible values

Any resulting stream structure, which would be fed to the printer/output device, must be "maximal" in the domain of streams. $\perp_S$ cannot occur anywhere within its result. This stipulation does not require that the result be isolated (e.g. the stream of ascending natural numbers is maximal, but not isolated.)

In addition, it is desirable to exclude values that contain an infinite series of left parentheses (assuming the printer makes a preorder traversal). The following definition identifies values that contain an infinite unbroken sequence of left parentheses.

**Definition:** A stream $s$ is *head-infinite* if preorder traversal of $s$ requires traversal of an infinite number of head fields, without any tail fields, in some sub-stream of $s$.

Suppose a preorder traversal of a stream outputs an 'H' every time it traverses a list **car**, a 'T' every time it traverses a list **cdr**, and an 'A' each time it finds an atom. The traversal of a head-infinite stream would eventually produce an infinite sequence of 'H's.

**Definition:** A stream is *head-finite* if it is not head-infinite.

A stream that is head-finite contains no infinite sequence of left parentheses; if $\perp_S$ doesn't occur in the stream, then the printer must eventually make progress and produce output during its traversal of the value produced by interpretation of the compiler's object code.

**Definition:** An *admissible* value is maximal in **S** and head-finite.

This definition permits certain kinds of infinite lists to appear in the head of a list, unlike the previous definition [12] which, as Wadler [39] pointed out, excludes them!

**Theorem 2.1:** The compiler preserves meaning.

**Outline of proof:** Strictness patterns are propagated in a leftmost-outermost manner, so that by structural induction over the compiler rules a strictness pattern will be compiled with the appropriate expression. ■

*Corollary: In its preorder traversal of a program's result, a printer proceeds exactly as far interpreting source code as interpreting object code when it prints an admissible value.*

### 2.7.4: Termination

Strictness patterns can be represented by graphs of list structures suggested by their notation (angle brackets become parentheses, $\perp_P$ and $ become distinguished tokens). All finite patterns can thus be represented by finite graphs. Infinite *cyclic* patterns contain only finite patterns and one or more infinite repetitions of either a finite pattern or a cyclic pattern. The repetitions within these patterns can also be represented by a cyclic graph containing a pointer to the structure representing the repetition. Infinite *acyclic* patterns contain at least one infinite pattern that is not cyclic, and so cannot be represented by either finite or cyclic graphs.

**Definition:** *Rational* strictness patterns are those patterns that can be represented by finite graphs.

**Lemma 2.2.1:** The compiler propagates only rational patterns.

**Proof** by structural induction on the compiler rules. There exists a finite representation for the initial inherited pattern, $\pi_0$. Rules 1-4, 6-8, and 10-12 propagate a pattern that is at most finitely increased in length. Rule 5 is distributive, and thus propagates a pattern no longer than that it receives. Rule 9 defines *rec-p* both as the pattern to be finally synthesized by the compilation of the *fix* expression and as the synthesized pattern to be passed up during compilation of a recursive call. Thus *rec-p* may be written as the *join* (or *meet*) of *rec-p* (inherited by the local lambda variable during compilation of a recursive call) and a pattern $\pi$ inherited by the other instances of the local lambda variable within the *fix* expression, or the rational expression

$$fix\,\lambda\ rec\text{-}p.\ \pi \sqcup rec\text{-}p\ .$$

□

**Lemma 2.2.2:** The compiler executes a finite number of rules.

**Proof:** Rules 1 through 10 recursively invoke the compiler on proper subexpressions or not at all. A simple induction on the structure of the expression shows that it terminates in a finite number of steps. Rules 11 and 12 decrement the resource to limit the possible expansion of the source code. Thus the compiler applies the rules finitely often. □

**Theorem 2.2:** The compiler terminates.

**Proof:** Finite cyclic graphs may be compared or combined in finite time. Thus, *meet*, *join*, and environment-lookup all terminate. By Lemmas 2.2.1 and 2.2.2, the compiler terminates. ■

## Chapter 3: $C$ is safe with respect to an instrumented interpreter $I$

This chapter shows that $C$ is safe relative to a series of rules specifying the behavior of an interpreter $I$ which is strict and which derives its lazy semantics from a lazy **cons** [11].

### Section 3.1: Outline of approach

An abstract machine, or instrumented interpreter (composed with a printer), is postulated that associates the interpretation of an expression with a stream of patterns representing the successive demand made upon its value, displaying those patterns as it interprets the code. The compilation of each piece of code is then proven to inherit a pattern no higher in **P** than the one displayed as the code was interpreted. For example, on page 52, the interpretation of equation (1) provides a trace of the evaluation of a simple expression; the second column is the stream of patterns displayed. In many cases, the pattern propagated by the compiler is far below that displayed by the interpreter because all cyclic patterns are forced to be no higher than the printer pattern $\pi_0$ in the lattice of strictness patterns. In fact, the equations presented here are even less powerful than they might otherwise be because *all* patterns inherited by identifiers, not just cyclic patterns, are bounded by the printer pattern for the sake of simplicity. This bound can easily be relaxed in future compilers.

The proof is a straightforward structural induction over the compiler rules, complicated only by the interpreter axioms, presented in Section 3.3.2, the invariants needed to constrain the compiler environment, and an extra proof and lemma needed to show that the compiler handles recursive function applications correctly. The lemma requires that the compiler be a continuous functional on *CEXP*, a special domain of syntactic expressions. *CEXP* contains only the finite expressions specified by the grammar *EXP* , except that it uses the least upper bound of an infinite chain of code *unfoldings* to represent a recursive function definition, rather than the finite *fix* expression presented in Chapter 2. However, the

structural induction deals only with finite approximations to the infinite program represented by this least upper bound, not to the non-isolated bound itself.

The following notation is used in several sections of this chapter.

— If a syntactic expression $[\![\mathtt{e}']\!]$ is a subexpression of another expression $[\![\mathtt{e}]\!]$, then this relationship is denoted as $[\![\mathtt{e}']\!] \in [\![\mathtt{e}]\!]$.

— $[\![\mathtt{e}]\!]\,[n]$ is the $n$th character in the string of characters, excluding strictness marks, that form $[\![\mathtt{e}]\!]$. Strictness marks are decorations on characters, and cannot be indexed. The predicate $(\mathit{Marked?}\ [\![\mathtt{e}]\!]\,[n])$ indicates whether a character is decorated with a strictness mark.

— Strictness patterns may be partitioned into two sets— those which contain strict marks and those which do not. $(\$ \in \alpha{\cdot}\pi)$ identifies patterns that do contain a strictness mark.

$\bot_S \in x$ denotes a value that either is $\bot_S$, or contains $\bot_S$ as a component. In other words, $x$ *contains* $\bot_S \equiv \bot_S \in x$.

### Section 3.2: $C$ is monotonic and continuous

*CEXP* is a domain of syntactic expressions with the following structure:

— Target code has at least as many strictness marks as source code, so it is no lower than the corresponding source code, and

— A recursive function definition is represented as a chain of successive finite unfoldings of that expression.

The function $U$ producing successive unfoldings of a recursive function definition is defined as

$$U : INT \times EXP \longrightarrow EXP$$

where

$U\ 0\ [\![\texttt{(fix:[f }\lambda\texttt{id. body])}]\!] = [\![\lambda\texttt{id. bottom}]\!]$.

$U\ n+1\ [\![$(fix:[f λid. body])$]\!] = [\![$λid. body′$]\!]$
**where** $[\![$body$]\!]' = [\![$body$]\!][(U\ n\ [\![$(fix:[f λid. body])$]\!])/[\![$f$]\!]]$.

For example,
$U\ 1\ [\![$(fix:[f λ1. <head:1 . f:tail:1>])$]\!] =$
$[\![$λ1. <head:1 . (λ1. bottom):tail:1>$]\!]$

$U\ 2\ [\![$(fix:[f λ1. <head:1 . f:tail:1>])$]\!] =$
$[\![$λ1. <head:1 . (λ1. <head:1 . (λ1. bottom):tail:1>):tail:1>$]\!]$

$U\ n\ [\![$(fix:[f λid. body])$]\!]$ is frequently written as

$$U_n\ [\![\texttt{(fix:[f λid. body])}]\!].$$

More formally, *CEXP* is ordered as follows:

$\forall\ [\![$E$]\!], [\![$E′$]\!] \in EXP - \{[\![$(fix :[id1 λ id2. e1]):e2$]\!]\}$
$[\![$E$]\!] \sqsubseteq [\![$E′$]\!] \iff$
$([\![$E$]\!][\epsilon/\$] = [\![$E′$]\!][\epsilon/\$])$ &
$(\forall\ s\ \textit{such that}\ 0 < s \leq |[\![$E$]\!]|, (Marked?\ [\![$E$]\!][n]) \implies (Marked?\ [\![$E′$]\!][n])$

∨

$\exists\ [\![$exp$]\!], j, k\ \textit{such that}\ [\![$exp$]\!] \in EXP,\ 0 \leq j \leq k,$
$[\![$E$]\!] = [\![(U_j\ [\![$exp$]\!])$:e2$]\!]$ *and* $[\![$E′$]\!] = [\![(U_k\ [\![$exp$]\!])$:e2$]\!]$

(A similar unfolding function can be defined for data-recursive definitions, but since such expressions do not yield synthesized patterns, they are not interesting enough to be discussed further here).

**Theorem 3.1:** $C : D \longrightarrow D$ is monotonic and continuous.

**Proof:** $D$ is now defined as $CEXP \times PAT \times ENV \times INT$ . $BEXP$, which contains syntactic expressions bound to a particular variable during compilation, is redefined to contain the least upper bound of a chain of approximations to a recursive function definition, rather than the finite *fix* expression given in Chapter 2.

$C$ is shown to be monotonic and continuous on each of $C$'s four arguments, if the other arguments are held constant.

**Lemma 3.1.1:** $C$ is monotonic and continuous in its first argument.

**Proof:** The compiler adds or verifies the same number of strictness marks when it is given two expressions which are identical, except that one contains strictness marks not present in the other. An approximation contains no more strictness marks, when compiled, than a higher approximation. Thus, the compiler preserves the relative positions of elements in its domain, and so is monotonic.
$\forall\, i > 0,\ (C\ [\![\mathbf{e}]\!]_{\,i}\ \alpha{\cdot}\pi\ \rho\ \iota)\downarrow 1 \sqsubseteq\ (C\ [\![(\sqcup_{i=1}^{\infty}\ [\![\mathbf{e}]\!]_{\,i})]\!]\ \ \alpha{\cdot}\pi\ \rho\ \iota)\downarrow 1$
as $C$ is monotonic in its first argument. Thus
$\bigsqcup_{i=1}^{\infty}\ ((C\ [\![\mathbf{e}]\!]_{\,i}\ \alpha{\cdot}\pi\ \rho\ \iota)\downarrow 1) \sqsubseteq (C\ [\![(\sqcup_{i=1}^{\infty}\ [\![\mathbf{e}]\!]_{\,i})]\!]\ \ \alpha{\cdot}\pi\ \rho\ \iota)\downarrow 1,$
as $\bigsqcup_{i=1}^{\infty}\ ((C\ [\![\mathbf{e}]\!]_{i}\ \alpha{\cdot}\pi\ \rho\ \iota)\downarrow 1)$ is the least upper bound for all the $(C\ [\![\mathbf{e}]\!]_{i}\ \alpha{\cdot}\pi\ \rho\ \iota)\downarrow 1$, while $(C\ [\![(\sqcup_{i=1}^{\infty}\ [\![\mathbf{e}]\!]_{\,i})]\!]\ \ \alpha{\cdot}\pi\ \rho\ \iota)\downarrow 1$ is an upper bound.

$\sqcup_{i=1}^{\infty}\ [\![\mathbf{e}]\!]_{\,i} \sqsubseteq \bigsqcup_{i=1}^{\infty}(C\ [\![\mathbf{e}]\!]_{\,i}\ \alpha{\cdot}\pi\ \rho\ \iota)\downarrow 1$, as $C$ does not remove strictness marks.
$(C\ [\![(\sqcup_{i=1}^{\infty}\ [\![\mathbf{e}]\!]_{\,i})]\!]\ \ \alpha{\cdot}\pi\ \rho\ \iota)\downarrow 1 \sqsubseteq\ (C(\bigsqcup_{i=1}^{\infty}((C\ [\![\mathbf{e}]\!]_{\,i}\ \alpha{\cdot}\pi\ \rho\ \iota)\downarrow 1))\ \alpha{\cdot}\pi\ \rho\ \iota)\downarrow 1,$
as $C$ is monotonic in its first argument. $\bigsqcup_{i=1}^{\infty}((C\ [\![\mathbf{e}]\!]_{\,i}\ \alpha{\cdot}\pi\ \rho\ \iota)\downarrow 1)$ is a fixed point for $C$'s first argument, since the compiler does not derive any new information from the marked code that would allow it to introduce a new strictness mark or unfolding, and simply verifies the marks already present (strictness marks are idempotent in compiled expressions as well as strictness patterns). Thus
$(C\ [\![(\sqcup_{i=1}^{\infty}\ [\![\mathbf{e}]\!]_{\,i})]\!]\ \ \alpha{\cdot}\pi\ \rho\ \iota)\downarrow 1 \sqsubseteq \bigsqcup_{i=1}^{\infty}\ ((C\ [\![\mathbf{e}]\!]_{\,i}\ \alpha{\cdot}\pi\ \rho\ \iota)\downarrow 1)$
□

**Lemma 3.1.2:** $C$ is monotonic and continuous in its second argument.

**Proof:** $\forall$ $[\![\mathbf{exp}]\!]$ $\alpha{\cdot}\pi$ $\rho$ $\iota$, $(C\ [\![\mathbf{exp}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota)\downarrow 2\downarrow 1 = \alpha{\cdot}\pi$,
so $C$ is trivially monotonic and continuous in its second argument. □

**Lemma 3.1.3:** $C$ is monotonic and continuous in its third argument.

**Proof:** It is assumed that the environments being considered are consistent with the expressions being compiled; for example, they are defined upon all free variables in $[\![\mathbf{e}]\!]$ and their definition is consistent with their use.

The domains of integers and tags stored in environment tuples are flat. The infinite chain representing a recursive function definition is only copied, and so does not change during a particular compilation. Therefore, the only interesting ordering is that produced by strictness patterns.

The compiler successively combines a fixed set of patterns determined by the compilation of $[\![\mathbf{e}]\!]$ with $\alpha{\cdot}\pi$, with the inherited patterns for a given variable in some $\rho_i$ and $\rho_{i+1}$, $\rho_i \sqsubseteq \rho_{i+1}$, in a chain of $i$ environments thus preserving their relative ordering. (Patterns may also depend upon other inherited patterns present in an environment, but this still preserves the ordering.) A similar argument shows that the relationship between environments ordered by the pattern function, *pa*, is also preserved.

The function *pa* is simply a bookkeeping mechanism necessary to a compiler that processes a finite representation of a *fix* expression; since the compiler discussed here processes an infinite piece of code, only patterns inherited by lambda bound variables need now be considered.

$(C\ [\![\mathbf{e}]\!]\ \alpha{\cdot}\pi\ (\sqcup_{i=1}^{\infty}\rho_i)\ \iota)\downarrow 2\downarrow 2\downarrow 1 \sqsubseteq \bigsqcup_{i=1}^{\infty}\ ((C\ [\![\mathbf{e}]\!]\ \alpha{\cdot}\pi\ \rho_i\ \iota)\downarrow 2\downarrow 2\downarrow 1)$
by an argument similar to that of Lemma 3.1.1.

$\bigsqcup_{i=1}^{\infty}\ ((C\ [\![\mathbf{e}]\!]\ \alpha{\cdot}\pi\ \rho_i\ \iota)\downarrow 2\downarrow 2\downarrow 1) \sqsubseteq (C\ [\![\mathbf{e}]\!]\ \alpha{\cdot}\pi\ (\sqcup_{i=1}^{\infty}\rho_i)\ \iota)\downarrow 2\downarrow 2\downarrow 1$
by the following argument, initially similar to that of Lemma 3.1.1, since $C$ only maintains or adds information to an environment. To see that $\bigsqcup_{i=1}^{\infty}((C\ [\![\mathbf{e}]\!]\ \alpha{\cdot}\pi\ \rho_i\ \iota)\downarrow 2\downarrow 2\downarrow 1)$ is a fixed point for $C$, note again that the compiler only maintains or adds information to an environment, so it is only

necessary to show that the compiler does not produce an environment higher after compiling an expression with $\bigsqcup_{i=1}^{\infty}((C\ [\![\mathtt{e}]\!]\ \alpha{\cdot}\pi\ \rho_i\ \iota){\downarrow}2{\downarrow}2{\downarrow}1)$. The only way in which the compiler can add new information to this environment is by propagating $\alpha{\cdot}\pi$ through the analysis of $[\![\mathtt{e}]\!]$. Let $a$ be the combination of *meets* and *joins* specified by the compiler rules of all the patterns inherited by instances of a given variable, $[\![\mathtt{id}]\!]$, during the compilation of $[\![\mathtt{e}]\!]$ with $\alpha{\cdot}\pi$.

Let the pattern already initially present in $\rho_i$, $(\rho_i\ [\![\mathtt{id}]\!]){\downarrow}2{\downarrow}1$, be $b_i$. Then $(((C\ [\![\mathtt{e}]\!]\ \alpha{\cdot}\pi\ \rho_i\ \iota){\downarrow}2{\downarrow}2{\downarrow}1)\ [\![\mathtt{id}]\!]){\downarrow}2{\downarrow}1$ is $a \sqcup b_i$. Since $a \sqcup \sqcup_{i=1}^{\infty} b_i = \bigsqcup_{i=1}^{\infty} a \sqcup b_i$, $C$ does not add more information to $\bigsqcup_{i=1}^{\infty}((C\ [\![\mathtt{e}]\!]\ \alpha{\cdot}\pi\ \rho_i\ \iota){\downarrow}2{\downarrow}2{\downarrow}1)$. □

**Lemma 3.1.4:** $C$ is monotonic and continuous in its fourth argument.

**Proof:** $\forall\ [\![\mathtt{exp}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota$, $(C\ [\![\mathtt{exp}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota){\downarrow}2{\downarrow}2{\downarrow}2{\downarrow}1 = \iota$,
so $C$ is trivially monotonic and continuous in its fourth argument. □
By Lemmas 3.1.1-4, $C : D \longrightarrow D$ is monotonic and continuous, proving Theorem 3.1. ■

It is interesting to note that this theorem holds even when there is no finite resource controlling the number of versions created, and irrational as well as rational patterns are propagated; of course, under these conditions the compiler might not terminate.

## Section 3.3: $I$ — an interpreter that displays demand patterns

Strictness patterns were introduced earlier so that the compiler could propagate strictness information approximating the behavior of the interpreter. Here, axioms describing part of the behavior of the interpreter itself are presented. This interpreter produces an extra result that describes the strictness pattern it used to evaluate an expression; this result can be regarded as a window through which various strictness patterns can be seen as they move by during the interpretation process.

The domain equation for this modified interpreter is

$I: CEXP \times IENV \longrightarrow VAL \times \mathbf{P}$
where
$\alpha_d \cdot \pi_d \in \mathbf{P}$
$R \in IENV = ID \longrightarrow [VAL + unbound]$

Note that the interpreter axioms are written as if its domain equation specified the displayed pattern as an argument, as in
$I: CEXP \times \mathbf{P} \times IENV \longrightarrow VAL$

The following example demonstrates the behavior of $I$:

When interpreting the program

```
add:<head:<1 . 2> 3>
```
(1)

$I$ moves through steps (i) through (viii) during its interpretation.

| | | | |
|---|---|---|---|
| $(i)$ | $I[\![$`add:<head:<1 . bottom> 3>`$]\!]$ | $fix\,\lambda\pi.\$\langle\pi\ ,\ \pi\rangle$ | $R$ |
| $(ii)$ | $I[\![$`<head:<1 . bottom> 3>`$]\!]$ | $\$\langle\$\bot\ ,\$\langle\$\bot\ ,\bot\rangle\rangle$ | $R$ |
| $(iii)$ | $I[\![$`head:<1 . bottom>`$]\!]$ | $\$\bot$ | $R$ |
| $(iv)$ | $I[\![$`<1 . bottom>`$]\!]$ | $\$\langle\$\bot\ ,\bot\rangle$ | $R$ |
| $(v)$ | $I[\![$`1`$]\!]$ | $\$\bot$ | $R$ |
| $(vi)$ | $I[\![$`bottom`$]\!]$ | $\bot$ | $R$ |
| $(vii)$ | $I[\![$`<3>`$]\!]$ | $\$\langle\$\bot\ ,\bot\rangle$ | $R$ |
| $(viii)$ | $I[\![$`3`$]\!]$ | $\$\bot$ | $R$ |

### 3.3.1: Notation

Some notation is required when it is necessary to discuss the patterns displayed by $I$ during a series of steps in the interpretive process, or when a pattern displayed during the interpretation of a particular expression must be identified. For example, it is useful to be able to discuss the pattern displayed by the interpretation of $[\![$`<1 . bottom>`$]\!]$ when the entire program being interpreted is (1). This will be denoted as:

$$[\![\texttt{<1 . bottom>}]\!] \blacksquare I[\![\texttt{add:<head:<1 . bottom> 3>}]\!]\ \alpha_d{\cdot}\pi_d\ R.$$

The corresponding compiler pattern is written as

$$[\![\texttt{<1 . bottom>}]\!] \bullet C\,[\![\texttt{add:<head:<1 . bottom> 3>}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota.$$

The compilation of a lambda expression often involves the creation of a series of environments, each with a possibly updated entry for the bound variable. In fact, there are at least $n$ new environments created, where $n$ is the number of references to $[\![\texttt{id}]\!]$ in the body of the expression. Occasionally, it will be necessary to discuss the pattern inserted in the environment entry created when the compiler has just seen the $i$th instance of this bound variable. This is written as:

$$(\,[\![\texttt{id}]\!]_i \ \textit{where}\ 1 \le i \le n) \bullet C\,[\![\texttt{exp}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota.$$

### 3.3.2: Interpreter axioms

In general, these axioms relate $I$'s behavior on expressions to its behavior on sub-expressions, rather than specify $I$ completely. $I$ may display one of several possible patterns when interpreting a piece of code, even though the resulting value will be probed in the same way. For example, the interpretation of the expression $[\![\texttt{<1 . 2>}]\!]$ could display the pattern $\$\langle\$\bot\ ,\ \$\bot\rangle$, or the pattern $\$\langle\pi_0\ ,\ \pi_0\rangle$; it is assumed to make no difference here, since 1 and 2 are atomic.

$$\forall\ [\![\texttt{e}]\!],\ \alpha_d{\cdot}\pi_d,\ R, \qquad \text{(a1)}$$
$$(\bot_S \notin I[\![\texttt{e}]\!]\ \alpha_d{\cdot}\pi_d\ R)\ \&\ (\$ \in \alpha_d{\cdot}\pi_d) \implies$$
$$(I[\![\texttt{e}]\!]\ \alpha_d{\cdot}\pi_d\ R\ =\ I[\![\$\texttt{e}]\!]\ \alpha_d{\cdot}\pi_d\ R).$$

**Explanation:** The interpreter is strict in the outer structure of any value containing some inner structure in which it is also strict, since it must traverse the outer structures to reach the inner ones. For this reason, for all patterns containing a strictness mark, the interpreter produces the same result whether or not the pattern is prefixed by a strictness mark. Since this is the case, the interpreter will require the outer structure of the resulting value, and so its behavior will not change when the expression producing the value is evaluated early.

The following five axioms are closely related to the correspondingly numbered equations in Chapter 2.

$$\forall\ [\![\mathtt{E}]\!],\ \alpha_d{\cdot}\pi_d,\ R,\ \textit{such that}\ [\![\mathtt{E}]\!] = [\![\mathtt{head:e}]\!], \qquad \text{(a3)}$$
$$(\bot_S \notin I[\![\mathtt{head:e}]\!]\ \alpha_d{\cdot}\pi_d\ R)\ \&\ (\$ \in \alpha_d{\cdot}\pi_d) \Longrightarrow$$
$$(I[\![\mathtt{head:e}]\!]\ \alpha_d{\cdot}\pi_d\ R = I[\![\mathtt{e}]\!]\ \${\cdot}\langle \${\cdot}\pi_d, \bot\rangle\ R).$$

$$\forall\ [\![\mathtt{E}]\!],\ \alpha_d{\cdot}\pi_d,\ R,\ \textit{such that}\ [\![\mathtt{E}]\!] = [\![\mathtt{tail:e}]\!], \qquad \text{(a4)}$$
$$(\bot_S \notin I[\![\mathtt{tail:e}]\!]\ \alpha_d{\cdot}\pi_d\ R)\ \&\ (\$ \in \alpha_d{\cdot}\pi_d) \Longrightarrow$$
$$(I[\![\mathtt{tail:e}]\!]\ \alpha_d{\cdot}\pi_d\ R = I[\![\mathtt{e}]\!]\ \${\cdot}\langle \bot, \${\cdot}\pi_d\rangle\ R).$$

$$\forall\ [\![\mathtt{E}]\!],\ \alpha_d,\ \alpha_1{\cdot}\pi_1,\ \alpha_2{\cdot}\pi_2,\ R,\ \textit{such that}\ [\![\mathtt{E}]\!] = [\![\mathtt{<e1\ .\ e2>}]\!], \qquad \text{(a5)}$$
$$(\bot_S \notin I[\![\mathtt{<e1\ .\ e2>}]\!]\ \alpha_d{\cdot}\langle\alpha_1{\cdot}\pi_1, \alpha_2{\cdot}\pi_2\rangle\ R) \Longleftrightarrow$$
$$(\bot_S \notin I[\![\mathtt{e1}]\!]\ \alpha_1{\cdot}\pi_1\ R)\ \&\ (\bot_S \notin I[\![\mathtt{e2}]\!]\ \alpha_2{\cdot}\pi_2\ R).$$

$$\forall\ [\![\mathtt{E}]\!],\ \alpha_d{\cdot}\pi_d,\ R,\ \textit{such that}\ [\![\mathtt{E}]\!] = [\![\mathtt{prim:<e1\ e2>}]\!], \qquad \text{(a6)}$$
$$(\bot_S \notin I[\![\mathtt{prim:<e1\ e2>}]\!]\ \alpha_d{\cdot}\pi_d\ R)\ \&\ (\$ \in \alpha_d{\cdot}\pi_d)$$
$$\Longrightarrow (\bot_S \notin I[\![\mathtt{<e1\ e2>}]\!]\ \$\langle\$\bot, \$\langle\$\bot, \bot\rangle\rangle R).$$

$$\forall\ [\![\mathtt{E}]\!],\ \alpha_d{\cdot}\pi_d,\ R,\ \textit{such that}\ [\![\mathtt{E}]\!] = [\![\mathtt{if:<e1\ e2\ e3>}]\!],\quad \text{(a7)}$$
$$(\perp_S \notin I[\![\mathtt{if:<e1\ e2\ e3>}]\!]\ \alpha_d{\cdot}\pi_d\ R)\ \&\ (\$ \in \alpha_d{\cdot}\pi_d) \Longrightarrow$$
$$(\perp_S \notin I[\![\mathtt{e1}]\!]\ \$\perp\ R)\ \&$$
$$(((I[\![\mathtt{e1}]\!]\ \$\perp\ R \neq \mathit{false})\ \&\ (\perp_S \notin I[\![\mathtt{e2}]\!]\ \alpha_d{\cdot}\pi_d\ R))$$
$$\vee$$
$$((I[\![\mathtt{e1}]\!]\ \$\perp\ R = \mathit{false})\ \&\ (\perp_S \notin I[\![\mathtt{e3}]\!]\ \alpha_d{\cdot}\pi_d\ R))).$$

$$\forall\ [\![\mathtt{E}]\!],\ \alpha_d{\cdot}\pi_d,\ R,\ \textit{such that}\ [\![\mathtt{E}]\!] = [\![\mathtt{(\lambda id.\ body):e}]\!],\quad \text{(a8)}$$
$$(\perp_S \notin I[\![\mathtt{(\lambda id.\ body):e}]\!]\ \alpha_d{\cdot}\pi_d\ R) \Longrightarrow$$
$$(\perp_S \notin I[\![\mathtt{body}]\!]\ \alpha_d{\cdot}\pi_d\ R[\,[\![\mathtt{e}]\!]\,/\,[\![\mathtt{id}]\!]\,])$$
$$\&\ (\perp_S \notin I[\![\mathtt{e}]\!]\ (\sqcup_{i=1}^{n}\ ([\![\mathtt{id}]\!]\ {}_i{\blacksquare} I[\![\mathtt{(\lambda id.\ body):e}]\!]\ \alpha_d{\cdot}\pi_d\ R))\ R).$$

**Explanation:** If the interpretation of a lambda application doesn't contain $\perp_S$ then the interpretation of the lambda body doesn't contain $\perp_S$ and it displays the original pattern, $\alpha_d{\cdot}\pi_d$. The interpreter can be thought of operationally as displaying the appropriate pattern each time it evaluates an instance of the bound variable, for which $[\![\mathtt{e}]\!]$ is substituted. The patterns displayed during this distribution of the evaluation of $[\![\mathtt{e}]\!]$ can also be regarded as one pattern displayed during only one evaluation of $[\![\mathtt{e}]\!]$—this one pattern is the least upper bound of those displayed when the interpreter evaluated the instances of the bound variable within the lambda body.

$$\forall\ [\![\mathtt{E}]\!],\ \alpha_d{\cdot}\pi_d,\ R,\ \textit{such that}\ [\![\mathtt{E}]\!] = [\![\mathtt{(fix:[f\ \lambda id.\ body]):e}]\!],\quad \text{(a9)}$$
$$(\perp_S \notin I[\![\mathtt{(fix:[f\ \lambda id.\ body]):e}]\!]\ \alpha_d{\cdot}\pi_d\ R) \Longrightarrow$$
$$(\perp_S \notin I[\![\mathtt{body}]\!]\ \alpha_d{\cdot}\pi_d\ R[\,[\![\mathtt{(fix:[f\ \lambda id.\ body])}]\!]\,/\,[\![\mathtt{f}]\!]\,])\ \&$$
$$(\perp_S \notin I[\![\mathtt{e}]\!]\ ([\![\mathtt{e}]\!]\ {\blacksquare} I[\![(\sqcup_{u=1}^{\infty}(U_u\ \mathtt{(fix:[f\ \lambda id.\ body])}))\mathtt{:e}]\!]\ \alpha_d{\cdot}\pi_d\ R)\ R).$$

**Explanation:** If the interpretation of a *fix* application doesn't contain $\perp_S$ then the interpretation of the lambda body doesn't contain $\perp_S$ and it displays the

original pattern, $\alpha_d \cdot \pi_d$. In addition, the interpretation of the application argument doesn't contain $\perp_S$ and it displays the least upper bound of all of the patterns displayed when the interpreter evaluates a possibly infinite number of unfoldings of the fix expression. *This pattern may be higher than the pattern actually displayed by the interpreter during any particular application evaluation.* However, it is reasonable to assume that the interpreter displays this pattern after considering the following argument:

> Assume that an oracle determines the precise number of applications of $A$ needed by the interpreter, and consider the pattern displayed by the interpreter as it evaluates $[\![\texttt{e}]\!]$. The least upper bound presented in (a9) is equal or higher in $P$ than this pattern only because it includes patterns that have been inherited from applications of $U$ that were not interpreted in producing the final answer. This is distinct from the interpretation of expressions whose values weren't required, causing the interpreter to display $\perp$.

Another way of looking at the problem can be shown using the following example:

```
[[(fix:[f λlst.
   if:<nil?:lst
       nil
       <add:<1 head:lst> . f:tail:lst>>]):
<1 2 3>]]
```

The oracle would produce four unfoldings of $[\![\texttt{f}]\!]$, and the interpreter would display $\$\langle\$\perp, \$\langle\$\perp, \$\langle\$\perp, \perp\rangle\rangle\rangle$ when evaluating $[\![\texttt{<1 2 3>}]\!]$. If the argument were extended to $[\![\texttt{<1 2 3 4 5 6>}]\!]$, then $\$\langle\$\perp, \perp\rangle$ would become correspondingly extended to $\$\langle\$\perp, \$\langle\$\perp, \$\langle\$\perp, \$\langle\$\perp, \perp\rangle\rangle\rangle\rangle$. However, no harm is done if the interpreter is said to display this pattern when the argument was $[\![\texttt{<1 2 3>}]\!]$ because there is no more structure left to be made strict after the 3 has been seen.

$\forall$ $[\![\mathtt{E}]\!]$, $\alpha_d{\cdot}\pi_d$, $R$, *such that* $[\![\mathtt{E}]\!] = [\![\mathtt{fix{:}[f\ e]}]\!]$, (a10)
$(\perp_S \notin I[\![\mathtt{fix{:}[f\ e]}]\!]\ \alpha_d{\cdot}\pi_d\ R) \Longrightarrow$
$(\perp_S \notin I[\![\mathtt{e}]\!]\ \alpha_d{\cdot}\pi_d\ R[\,[\![\mathtt{fix{:}[f\ e]}]\!]\,/\,[\![\mathtt{f}]\!]\,])$.

**Explanation:** If the interpretation of a data recursion, or recursively defined data structure, doesn't contain $\perp_S$, then the interpretation of the structure itself doesn't contain $\perp_S$ and the pattern it displays is the same as the original.

$\forall$ $[\![\mathtt{E}]\!]$, $\alpha_d{\cdot}\pi_d$, $R$, *such that* $[\![\mathtt{E}]\!] = [\![\mathtt{f{:}e}]\!]$, (a11)
$(\perp_S \notin I[\![\mathtt{f{:}e}]\!]\ \alpha_d{\cdot}\pi_d\ R) \Longrightarrow$
$\exists$ $[\![\mathtt{E}]\!]$ *such that* $((R\ [\![\mathtt{f}]\!]) = [\![\mathtt{(fix{:}[f\ \lambda id.\ body])}]\!] = \mathtt{E})$ &
$(\perp_S \notin I[\![\mathtt{e}]\!]\ ([\![\mathtt{e}]\!] \blacksquare I[\![(\sqcup_{u=1}^{\infty}(U_u(\mathtt{fix{:}[f\ \lambda id.\ body]})))\mathtt{{:}e}]\!]\ \alpha_d{\cdot}\pi_d\ R)\ R)$.

**Explanation:** If the interpretation of a function application doesn't contain $\perp_S$, then the interpreter doesn't contain $\perp_S$ when evaluating the argument, and it then displays a pattern that is the least upper bound of the patterns displayed when the interpreter evaluates the lambda expression bound to **f**.

**Theorem 3.2:** *C* is safe with respect to *I*

The proof is a structural induction on the compiler rules, with an induction hypothesis which relates the interpretation of the code produced by the compiler to the interpretation of the source code.

The induction hypothesis is that interpretation of the compiler source code and compiler object code produce equal values when:

— interpretation of the source code produces a value that doesn't contain $\perp_S$

— the compile-time environment satisfies certain constraints, and

— the compilation inherits a pattern that is at most equal to the pattern displayed by the interpreter.

(Assume for the purposes of simplifying the proof that all variable names are unique.)

The formal inductive hypothesis is:

$$
\begin{array}{l}
\forall\ [\![\mathbf{exp}]\!],\ \alpha_d{\cdot}\pi_d,\ \alpha{\cdot}\pi,\ \rho,\ \iota,\ R,\ R', \\
\quad \textit{such that}\ [\![\mathbf{exp}]\!] \in \mathit{CEXP},\ \alpha{\cdot}\pi, \alpha_d{\cdot}\pi_d \in P, \rho \in \mathit{ENV}, \\
\qquad \iota \in \mathit{INT},\ R, R' \in \mathit{IENV}\ \textit{and}\ \mathit{Equivalently\text{-}extended}\ (\rho, R, R') \\
\\
\quad \textbf{let}\ [\![\mathbf{exp'}]\!]\ \alpha'{\cdot}\pi'\ \rho'\ \iota'\ \textbf{be}\ C\ [\![\mathbf{exp}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota\ \textbf{in} \\
\\
\quad \mathit{Safe\text{-}pat}\ (\alpha{\cdot}\pi, \alpha_d{\cdot}\pi_d)\ \&\ \mathit{Safe\text{-}comp\text{-}env}\ (\rho)\ \&\ (\bot_S \notin I[\![\mathbf{exp}]\!]\ \alpha_d{\cdot}\pi_d\ R) \\
\quad\ \Longrightarrow \\
\quad (I[\![\mathbf{exp}]\!]\ \alpha_d{\cdot}\pi_d\ R = I[\![\mathbf{exp'}]\!]\ \alpha_d{\cdot}\pi_d\ R')\ \&\ \mathit{Safe\text{-}comp\text{-}env}\ (\rho')
\end{array}
$$

In the following definitions, $dom\ f = \{[\![\mathtt{id}]\!] : ID \mid f\ [\![\mathtt{id}]\!] \neq unbound\}$. Cyclic patterns are non-isolated elements in the sub-lattice of finitely representable elements in $P$, meaning that they are limit points which are finitely representable.

$$
\begin{array}{l}
\mathit{Equivalently\text{-}extended}\ (\rho, R, R') \equiv \\
dom\ \rho = dom\ R = dom\ R' \\
\\
\mathit{Extension\text{-}of}\ (\rho, \rho') \equiv \\
dom\ \rho \supset dom\ \rho'
\end{array}
$$

$$
\begin{array}{l}
\textit{Safe-pat}\ (\alpha{\cdot}\pi, \alpha_d{\cdot}\pi_d) \equiv \\
(\textit{Cyclic}\ \alpha_d{\cdot}\pi_d \ \ \& \ \ \alpha{\cdot}\pi \sqsubseteq (\alpha_d{\cdot}\pi_d \sqcap \pi_0)) \\
\vee \\
(\neg \textit{Cyclic}\ \alpha_d{\cdot}\pi_d \ \ \& \ \ \alpha{\cdot}\pi \sqsubseteq \alpha_d{\cdot}\pi_d)
\end{array}
$$

Cyclic patterns propagated by the compiler must be no higher than the printer pattern; all patterns propagated by the compiler must be no higher than the corresponding pattern displayed by the interpreter.

$$
\begin{array}{l}
\textit{Safe-comp-env}\ (\rho) \equiv \\
\forall\ [\![\texttt{ide}]\!] \\
\quad (\rho\ [\![\texttt{ide}]\!]) = \textit{unbound} \vee \\
\quad \Big((\rho\ [\![\texttt{ide}]\!]) \neq \textit{unbound}\ \& \\
((\textit{Binding-type}(\rho\ [\![\texttt{ide}]\!]) = \textbf{lambda}) \implies \textit{Safe-lambda-exp}\ (\rho, [\![\texttt{ide}]\!])) \ \& \\
((\textit{Binding-type}(\rho\ [\![\texttt{ide}]\!]) = \textbf{fix}) \implies \textit{Safe-fix-exp}\ (\rho, [\![\texttt{ide}]\!]))\Big)
\end{array}
$$

Identifiers may be bound in either a *lambda* or a *fix* environment; the compiler forms each environment entry assuming only these cases exist. The syntactic expressions containing an identifier are $[\![$`(λid. body):e`$]\!]$, $[\![$`(fix:[f λid. body]):e`$]\!]$, $[\![$`fix:[id e]`$]\!]$, $[\![$`f:e`$]\!]$ and $[\![$`id`$]\!]$. An environment must be shown to be safe whenever the compiler creates a new identifier binding.

*Safe-lambda-exp* $(\rho, [\![\texttt{ide}]\!]) \equiv$

$\forall\ R, \alpha{\cdot}\pi, \alpha_d{\cdot}\pi_d, \iota, \rho'$
*such that* *Safe-pat* $(\alpha{\cdot}\pi, \alpha_d{\cdot}\pi_d)$,
*Equivalently-extended* $(\rho', R)$,
*and Extension-of* $(\rho, \rho')$

$\forall\ [\![\texttt{E}]\!]$ *such that* $[\![\texttt{E}]\!] = [\![(\lambda\texttt{ide.}\ \texttt{body}):\texttt{e}]\!]$

$pa \sqsubseteq ((\ [\![\texttt{ide}]\!]_n$ *where* $1 \leq i \leq n) \bullet C\ [\![\texttt{E}]\!]\ \alpha{\cdot}\pi\ \rho'\ \iota) \sqsubseteq$

$(\sqcup_{i=1}^{n}\ (\ [\![\texttt{ide}]\!]_i \blacksquare I [\![\texttt{E}]\!]\ \alpha_d{\cdot}\pi_d\ R))$

**where** $(\rho\ [\![\texttt{ide}]\!]) = \langle binding, pa, 0, b\text{-}type \rangle$

If the identifier $[\![\texttt{ide}]\!]$ is bound in a *lambda* environment, then the compiler environment created when the $n$th instance of $[\![\texttt{ide}]\!]$ was compiled during the compilation of $[\![\texttt{body}]\!]$ contains an inherited pattern which must be no higher than the pattern displayed by the interpreter when it interprets the application of the lambda expression to $[\![\texttt{e}]\!]$.

*Safe-fix-exp* $(\rho,\ [\![\texttt{ide}]\!]) \equiv$

$\forall\ R, \alpha{\cdot}\pi, \alpha_d{\cdot}\pi_d, \iota, \rho'$
*such that* *Safe-pat* $(\alpha{\cdot}\pi, \alpha_d{\cdot}\pi_d)$,
*Equivalently-extended* $(\rho', R)$,
*Extension-of* $(\rho, \rho')$

$\forall\ [\![\texttt{E}]\!]$ *such that* $[\![\texttt{E}]\!] = [\![\texttt{(fix:[ide exp]):e}]\!]$
$((pa\ \alpha{\cdot}\pi) \sqsubseteq [\![\texttt{e}]\!]\ {}_{\blacksquare}I[\![\texttt{E}]\!]\ \alpha_d{\cdot}\pi_d\ R)$
&

$\forall\ [\![\texttt{E}]\!]$ *such that* $[\![\texttt{E}]\!] = [\![\texttt{ide:e}']\!]$
$((pa\ \alpha{\cdot}\pi) \sqsubseteq [\![\texttt{e}']\!]\ {}_{\blacksquare}I[\![\texttt{E}]\!]\ \alpha_d{\cdot}\pi_d\ R)$
&

$\forall\ [\![\texttt{E}]\!]$ *such that* $[\![\texttt{E}]\!] = \{\ [\![\texttt{fix:[ide exp]}]\!]\ ,\ [\![\texttt{ide}]\!]\ \}$
$\alpha{\cdot}\pi \sqsubseteq \alpha_d{\cdot}\pi_d$

**where** $(\rho\ [\![\texttt{ide}]\!]) = \langle binding, pa, v\text{-}count, b\text{-}type\rangle$

If the identifier is bound in a *fix* environment, then it either represents a function or a data structure, depending upon whether it originally appeared in $[\![\texttt{(fix:[f λid. body]):e}]\!]$ or $[\![\texttt{fix:[id e]}]\!]$. If it represents a function $[\![\texttt{f}]\!]$, then for every safe pattern inherited by the compilation of an application of $[\![\texttt{f}]\!]$, the environment must be shown to contain a safe synthesized pattern that can be inherited by the compilation of the application argument. If the identifier represents a data structure, then no synthesized pattern is involved, and the given safe relationship between the compiler pattern $\alpha{\cdot}\pi$ and the interpreter's displayed pattern $\alpha_d{\cdot}\pi_d$ is enough to show that recursively defined data structures are safely compiled.

**Base case:**

Assume that the entire program being compiled does not represent a value that contains $\perp_S$. The compiler initially inherits the printer pattern, while the

interpreter initially displays $\mathit{fix}\,\lambda\pi.\$\langle\pi\ ,\ \pi\rangle$, a pattern which is above the printer pattern $\pi_0$ in $P$. The initial compiler and interpreter environments contain no entries.

$$C\,[\![\mathbf{const}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota = [\![\$\mathbf{const}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota. \qquad (\mathcal{C}\ 1)$$

**Case 1:** $[\![\mathbf{exp}]\!] = [\![\mathbf{const}]\!]$

**Given:** $\alpha_d{\cdot}\pi_d, R, R'$ *such that*

i) $\perp_S \notin I\,[\![\mathbf{const}]\!]\ \alpha_d{\cdot}\pi_d\ R$

ii) *Safe-pat* $(\alpha{\cdot}\pi, \alpha_d{\cdot}\pi_d)$

iii) *Safe-comp-env* $(\rho)$

iv) *Equivalently-extended* $(\rho, R, R')$

**Need to show:**

$(I\,[\![\mathbf{const}]\!]\ \alpha_d{\cdot}\pi_d\ R = I\,[\![\$\mathbf{const}]\!]\ \alpha_d{\cdot}\pi_d\ R')$ & *Safe-comp-env* $(\rho)$

**Proof:**

The interpretation of a constant always terminates, so marking a constant is always safe.

$(I\,[\![\mathbf{const}]\!]\ \alpha_d{\cdot}\pi_d\ R = I\,[\![\$\mathbf{const}]\!]\ \alpha_d{\cdot}\pi_d\ R')$ & *Safe-comp-env* $(\rho)$

By iii) and $(\mathcal{C}\ 1)$. □

$C\,[\![\mathtt{e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota$, **where** $(\$ \notin \alpha{\cdot}\pi) =$ (C 2)

$$
\begin{cases}
[\![\mathtt{e}[\,[\![\mathtt{fix:[id\ exp]}]\!]\,/\,[\![\mathtt{e'}]\!]\,]]\!]\ \alpha{\cdot}\pi\ \rho\ \iota \\
\textbf{where}\ [\![\mathtt{fix:[id\ exp]}]\!] = \\
\quad (\mathit{Binding}\ \ (\rho\ [\![\mathtt{e'}]\!])) \\
\qquad\qquad \text{if } \exists\ [\![\mathtt{e'}]\!] \in [\![\mathtt{e}]\!]\ \mathit{such\ that}\ [\![\mathtt{e'}]\!] \in \mathit{ID} \\
\qquad\qquad \&\ (\mathit{Binding\text{-}type}\ (\rho\ [\![\mathtt{e'}]\!])) = \textbf{fix}\ , \\
\\
[\![\mathtt{e}[\,[\![\mathtt{(fix:[f\ \lambda id.\ body])\!:\!exp}]\!]\,/\,[\![\mathtt{e'}]\!]\,]]\!]\ \alpha{\cdot}\pi\ \rho\ \iota \\
\textbf{where}\ [\![\mathtt{(fix:[f\ \lambda id.\ body])}]\!] = \\
\quad (\mathit{Binding}\ \ (\rho\ [\![\mathtt{f}]\!])) \\
\qquad\qquad \text{if } \exists\ [\![\mathtt{e'}]\!] \in [\![\mathtt{e}]\!]\ \mathit{such\ that}\ [\![\mathtt{e'}]\!] = [\![\mathtt{f:exp}]\!] \\
\qquad\qquad \&\ (\mathit{Binding\text{-}type}\ (\rho\ [\![\mathtt{f}]\!])) = \textbf{fix}\ , \\
\\
[\![\mathtt{e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota \\
\qquad\qquad \text{otherwise.}
\end{cases}
$$

**Case 2:** $[\![\mathtt{exp}]\!] = [\![\mathtt{e}]\!]$ *where* $(\$ \notin \alpha{\cdot}\pi)$

**Given:** $\alpha_d{\cdot}\pi_d, R, R'$ *such that*

i) $\perp_S \notin I\,[\![\mathtt{e}]\!]\ \alpha_d{\cdot}\pi_d\ R$

ii) *Safe-pat* $(\alpha{\cdot}\pi, \alpha_d{\cdot}\pi_d)$

iii) *Safe-comp-env* $(\rho)$

iv) *Equivalently-extended* $(\rho, RR')$

**Need to show each of A, B and C below,** which are exhaustive:

$(I[\![\mathtt{e}]\!]\ \alpha_d{\cdot}\pi_d\ R = I[\![\mathtt{e}[\,[\![\mathtt{fix:[id\ exp]}]\!]\,/\,[\![\mathtt{e'}]\!]\,]]\!]\ \alpha_d{\cdot}\pi_d\ R')$ & *Safe-comp-env* $(\rho)$(A)

if $\exists\ [\![\mathtt{e'}]\!] \in [\![\mathtt{e}]\!]$ *such that* $[\![\mathtt{e'}]\!] \in \mathit{ID}$

& $(\mathit{Binding\text{-}type}\ (\rho\ [\![\mathtt{e'}]\!])) = \textbf{fix}$

$(I\,[\![\mathtt{e}]\!]\ \alpha_d{\cdot}\pi_d\ R = I\,[\![\mathtt{e}[\,[\![\mathtt{(fix:[f\ \lambda id.\ body])\!:\!exp}]\!]\,/\,[\![\mathtt{e'}]\!]\,]]\!]\ \alpha_d{\cdot}\pi_d\ R')$ (B)

& *Safe-comp-env* $(\rho)$

if $\exists$ $[\![\mathtt{e}']\!] \in [\![\mathtt{e}]\!]$ *such that* $[\![\mathtt{e}']\!] = [\![\mathtt{f:exp}]\!]$
& (*Binding-type* $(\rho\ [\![\mathtt{f}]\!])) = \mathbf{fix}$ ,

$(I[\![\mathtt{e}]\!]\ \alpha_d{\cdot}\pi_d\ R = I[\![\mathtt{e}]\!]\ \alpha_d{\cdot}\pi_d\ R')$ & *Safe-comp-env* $(\rho)$ (C)
otherwise

**Case 2.A**

$(\rho\ [\![\mathtt{e}]\!])\downarrow 1\ =\ [\![\mathtt{fix:[f\ exp]}]\!]$ (1)

By ($\mathcal{C}$ 2).

*Equivalently-extended* $(\rho, R, R')$ (2)

By iv).

$(I[\![\mathtt{e}]\!]\ \alpha_d{\cdot}\pi_d\ R = I[\![\mathtt{e}[\,[\![\mathtt{fix:[id\ exp]}]\!] / [\![\mathtt{e}']\!]\,]]\!]\ \alpha_d{\cdot}\pi_d\ R')$
& *Safe-comp-env* $(\rho)$

By (1), (2), and iii).

**Case 2.B**

$(\rho\ [\![\mathtt{e}]\!])\downarrow 1\ =\ [\![\mathtt{(fix:[f\ \lambda id.\ body])}]\!]$ (1)

By ($\mathcal{C}$ 2).

*Equivalently-extended* $(\rho, R, R')$ (2)

By iv).

$(I[\![\mathtt{e}]\!]\ \alpha_d{\cdot}\pi_d\ R = I[\![\mathtt{e}[\,[\![\mathtt{(fix:[f\ \lambda id.\ body]):exp}]\!] / [\![\mathtt{e}']\!]\,]]\!]\ \alpha_d{\cdot}\pi_d\ R')$
& *Safe-comp-env* $(\rho)$

By (1), (2), substitution and iii).

**Case 2.C**

$(I[\![\mathtt{e}]\!]\ \alpha_d{\cdot}\pi_d\ R = I[\![\mathtt{e}]\!]\ \alpha_d{\cdot}\pi_d\ R')$ & *Safe-comp-env* $(\rho)$

By iii).

$C\ [\![\texttt{head:e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota = [\![\texttt{head:e}_1]\!]\ \alpha{\cdot}\pi\ \rho_1\ \iota$ ($\mathcal{C}$ 3)
**where** $[\![\texttt{e}_1]\!]\ \alpha_1{\cdot}\pi_1\ \rho_1\ \iota_1 = C\ [\![\texttt{e}]\!]\ \alpha{\cdot}\langle\alpha{\cdot}\pi,\ \perp\rangle\ \rho\ \iota.$

**Case 3:** $[\![\texttt{exp}]\!] = [\![\texttt{head:e}]\!]$

**Given:** $\alpha_d{\cdot}\pi_d, R, R'$ *such that*

i) $\perp_S \notin I\ [\![\texttt{head:e}]\!]\ \alpha_d{\cdot}\pi_d R$

ii) *Safe-pat* $(\alpha{\cdot}\pi, \alpha_d{\cdot}\pi_d)$

iii) *Safe-comp-env* $(\rho)$

iv) *Equivalently-extended* $(\rho, R, R')$

**Need to show:**

$(I\ [\![\texttt{head:e}]\!]\ \alpha_d{\cdot}\pi_d\ R\ = I\ [\![\texttt{head:e}_1]\!]\ \alpha_d{\cdot}\pi_d\ R')$ & *Safe-comp-env* $(\rho_1)$

**Proof:**

The interpreter and compiler patterns may or may not contain \$. However one case, $(\$ \notin \alpha_d{\cdot}\pi_d)$ & $(\$ \in \alpha{\cdot}\pi)$, is excluded by ii). Thus there are the following three other cases:

Case 3.1) $(\$ \notin \alpha_d{\cdot}\pi_d)$ & $(\$ \notin \alpha{\cdot}\pi)$ See Case 2).

Case 3.2) $(\$ \in \alpha_d{\cdot}\pi_d)$ & $(\$ \notin \alpha{\cdot}\pi)$ See Case 2).

Case 3.3) $(\$ \in \alpha_d{\cdot}\pi_d)$ & $(\$ \in \alpha{\cdot}\pi)$

Steps (1) through (5) justify the use of the induction hypothesis (IH) in (6). Steps (2) through (5) show that the corresponding substructures of the compiler and interpreter patterns in ($\mathcal{C}$ 2), line 2, are safe.

$\perp_S \notin I\ [\![\texttt{e}]\!]\ \$\langle\$\pi_d, \perp\rangle\ R$ (1)

By i) and a3).

*Safe-pat* $(\alpha{\cdot}\pi, \$\pi_d)$ (2)

By ii).

$Safe\text{-}pat\ (\bot, \bot)$ (3)

$Safe\text{-}pat\ (\langle \alpha \cdot \pi, \bot \rangle, \langle \$\pi_d, \bot \rangle)$ (4)

By (2), (3) and defn. of $P$.

$Safe\text{-}pat\ (\alpha \cdot \langle \alpha \cdot \pi, \bot \rangle, \$\langle \$\pi_d, \bot \rangle)$ (5)

By (4) and defn. of $P$.

$(I[\![\mathbf{e}]\!]\ \alpha_d \cdot \pi_d\ R = I[\![\mathbf{e_1}]\!]\ \alpha_d \cdot \pi_d\ R')\ \&\ Safe\text{-}comp\text{-}env\ (\rho_1)$. (6)

By (1), (5), iii) and IH.

$I[\![\mathbf{head:e}]\!]\ \alpha_d \cdot \pi_d\ R\ = I[\![\mathbf{head:e_1}]\!]\ \alpha_d \cdot \pi_d\ R'$

By (6) and substitution. □

$$C\,[\![\texttt{tail:e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota = [\![\texttt{tail:e}_1]\!]\ \alpha{\cdot}\pi\ \rho_1\ \iota \quad \textbf{where}\ [\![\texttt{e}_1]\!]\ \alpha_1{\cdot}\pi_1\ \rho_1\ \iota_1 = C\,[\![\texttt{e}]\!]\ \alpha{\cdot}\langle\perp,\ \alpha{\cdot}\pi\rangle\ \rho\ \iota. \qquad (\mathcal{C}\ 4)$$

**Case 4:** $[\![\texttt{exp}]\!] = [\![\texttt{tail:e}]\!]$

**Given:** $\alpha_d{\cdot}\pi_d, R, R'$ *such that*

i) $\perp_S \notin I\,[\![\texttt{tail:e}]\!]\ \alpha_d{\cdot}\pi_d R$

ii) *Safe-pat* $(\alpha{\cdot}\pi, \alpha_d{\cdot}\pi_d)$

iii) *Safe-comp-env* $(\rho)$

iv) *Equivalently-extended* $(\rho, R, R')$

**Need to show:**

$(I\,[\![\texttt{tail:e}]\!]\ \alpha_d{\cdot}\pi_d\ R = I\,[\![\texttt{tail:e}_1]\!]\ \alpha_d{\cdot}\pi_d\ R')$ & *Safe-comp-env* $(\rho_1)$

**Proof:**

Similar to that of Case 3. □

$$C\,[\![\texttt{<e1 . e2>}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota = \qquad (\mathcal{C}\ 5)$$

$$\begin{cases} C\,[\![\texttt{<e1 . e2>}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota, \textbf{where}\ (\$ \notin \alpha{\cdot}\pi) & \text{if } (\$ \notin \pi); \\ [\![\texttt{<}\alpha_1{\cdot}\texttt{e1}_1\ .\ \alpha_2{\cdot}\texttt{e2}_2\texttt{>}]\!]\ \alpha{\cdot}\pi\ \rho_2\ \iota, & \text{otherwise;} \end{cases}$$

**where**

$\alpha_1{\cdot}\pi_1 = (\pi{\downarrow}1)$

$\alpha_2{\cdot}\pi_2 = (\pi{\downarrow}2)$

$[\![\texttt{e1}_1]\!]\ \alpha_1{\cdot}\pi_1\ \rho_1\ \iota_1 = C\,[\![\texttt{e1}]\!]\,(\pi{\downarrow}1)\ \rho\ \iota;$

$[\![\texttt{e2}_2]\!]\ \alpha_2{\cdot}\pi_2\ \rho_2\ \iota_2 = C\,[\![\texttt{e2}]\!]\,(\pi{\downarrow}2)\ \rho_1\ \iota.$

**Case 5:** $[\![\texttt{exp}]\!] = [\![\texttt{<e1 . e2>}]\!]$

**Given:** $\alpha_d{\cdot}\pi_d, R, R'$ *such that*

i) $\perp_S \notin I\,[\![\texttt{<e1 . e2>}]\!]\ \alpha_d{\cdot}\pi_d R$

ii) *Safe-pat* $(\alpha{\cdot}\pi, \alpha_d{\cdot}\pi_d)$

iii) *Safe-comp-env* $(\rho)$

iv) *Equivalently-extended* $(\rho, R, R')$

**Need to show:**

Case 2) shows that the induction hypothesis is maintained when $(\$ \notin \pi)$. Otherwise, when $(\$ \in \alpha{\cdot}\pi)$, must show that:

$(I\,[\![\texttt{<e1 . e2>}]\!]\ \alpha_d{\cdot}\pi_d\ R = I\,[\![\texttt{<}\alpha_1{\cdot}\texttt{e1}^1\ .\ \alpha_2{\cdot}\texttt{e2}^2\texttt{>}]\!]\ \alpha_d{\cdot}\pi_d\ R')$

& *Safe-comp-env* $(\rho_2)$

**Proof:**

Since it is known that $(\$ \in \pi)$, it is also known that $(\$ \in \alpha{\cdot}\pi)$, and by ii), $(\$ \in \alpha_d{\cdot}\pi_d)$. Thus there is only one case:

Case 5.1) $(\$ \in \alpha_d{\cdot}\pi_d)$ & $(\$ \in \alpha{\cdot}\pi)$

Steps (1) and (2) justify the use of the induction hypothesis in (3). Steps (4) and (5) justify the use of the induction hypothesis in (6).

$\perp_S \notin I\,[\![\texttt{e1}]\!]\ \alpha_{d1}{\cdot}\pi_{d1}\ R$ (1)

By a5) and i).

$Safe\text{-}pat\ (\pi{\downarrow}1, \alpha_{d1}{\cdot}\pi_{d1})$ (2)

By ii)

$(I[\![\texttt{e1}]\!]\ \alpha_{d1}{\cdot}\pi_{d1}\ R\ = I[\![\texttt{e1}_1]\!]\ \alpha_{d1}{\cdot}\pi_{d1}\ R)$
$\quad \&\ \ Safe\text{-}comp\text{-}env\ (\rho_1)$ (3)

By (1), (2), iii) and IH.

$\perp_S \notin I[\![\texttt{e2}]\!]\ \alpha_{d2}{\cdot}\pi_{d2}\ R$ (4)

By a5) and i).

$Safe\text{-}pat\ (\pi{\downarrow}2, \alpha_{d2}{\cdot}\pi_{d2})$ (5)

By ii)

$(I[\![\texttt{e2}]\!]\ \alpha_{d2}{\cdot}\pi_{d2}\ R\ = I[\![\texttt{e2}_2]\!]\ \alpha_{d2}{\cdot}\pi_{d2}\ R')$
$\quad \&\ \ Safe\ (\rho_2)$ (6)

By (4), (5), (3) and IH.

$I[\![\texttt{<e1 . e2>}]\!]\ \alpha_d{\cdot}\langle\alpha_1{\cdot}\pi_1, \alpha_2{\cdot}\pi_2\rangle\ R\ =$
$I[\![\texttt{<}\alpha_1\texttt{·e1}_1\ \texttt{.}\ \alpha_2\texttt{·e2}_2\texttt{>}]\!]\ \alpha_d{\cdot}\langle\alpha_1{\cdot}\pi_1, \alpha_2{\cdot}\pi_2\rangle\ R'$

By (3), (6), substitution and a1). □

$C\ [\![\texttt{prim:<e1 e2>}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota = [\![\texttt{prim:e}_1]\!]\ \alpha{\cdot}\pi\ \rho_1\ \iota$ ($\mathcal{C}$ 6)
**where**
$[\![\texttt{e}_1]\!]\ \alpha_1{\cdot}\pi_1\ \rho_1\ \iota_1 = C\ [\![\texttt{<e1 e2>}]\!]\ \langle \$\bot\ , \$\langle\ \$\bot, \bot\rangle\rangle\ \rho\ \iota.$

**Case 6:** $[\![\texttt{exp}]\!]\ =\ [\![\texttt{arith-prim:<e1 e2>}]\!]$

**Given:** $\alpha_d{\cdot}\pi_d, R, R'$ *such that*

i) $\bot_S \notin I\ [\![\texttt{arith-prim:<e1 e2>}]\!]\ \alpha_d{\cdot}\pi_d R$

ii) *Safe-pat* $(\alpha{\cdot}\pi, \alpha_d{\cdot}\pi_d)$

iii) *Safe-comp-env* $(\rho)$

iv) *Equivalently-extended* $(\rho, R, R')$

**Need to show:**

$(I\ [\![\texttt{arith-prim:<e1 e2>}]\!]\ \alpha_d{\cdot}\pi_d\ R\ = I\ [\![\texttt{arith-prim:e}_1]\!]\ \alpha_d{\cdot}\pi_d\ R')$
& *Safe-comp-env* $(\rho_1)$

**Proof:**

As shown in Case 3), there are three possible cases.

Case 6.1) $(\$ \notin \alpha_d{\cdot}\pi_d)$ & $(\$ \notin \alpha{\cdot}\pi)$; See Case 2).

Case 6.2) $(\$ \in \alpha_d{\cdot}\pi_d)$ & $(\$ \notin \alpha{\cdot}\pi)$; See Case 2).

Case 6.3) $(\$ \in \alpha_d{\cdot}\pi_d)$ & $(\$ \in \alpha{\cdot}\pi)$

Steps (1) and (2) justify the use of the induction hypothesis in (3).

$\bot_S \notin I\ [\![\texttt{<e1 e2>}]\!]\ \$\langle\$\bot, \$\langle\$\bot, \bot\rangle\rangle\ R$ (1)

By i) and a6).

*Safe-pat* $(\langle\$\bot, \$\langle\$\bot, \bot\rangle\rangle, \$\langle\$\bot, \$\langle\$\bot, \bot\rangle\rangle)$ (2)

By defn. of P

$(I\ [\![\texttt{<e1 e2>}]\!]\ \alpha_d{\cdot}\pi_d\ R\ = I\ [\![\texttt{e}_1]\!]\ \alpha_d{\cdot}\pi_d\ R')$
& *Safe-comp-env* $(\rho_1)$ (3)

By (1), (2), iii) and IH.

$I\ [\![\texttt{arith-prim:<e1 e2>}]\!]\ \alpha_d{\cdot}\pi_d\ R\ = I\ [\![\texttt{arith-prim:e}_1]\!]\ \alpha_d{\cdot}\pi_d\ R'$

By (3) and substitution. □

(C 7)

$C\,[\![\texttt{if:<e1 e2 e3>}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota$
$\quad = [\![\texttt{if:<\$e1}_1\ \texttt{e2}_2\ \texttt{e3}_3\texttt{>}]\!]\ \alpha{\cdot}\pi\ \rho_4\ \iota$

**where**

$[\![\texttt{e1}_1]\!]\ \alpha_1{\cdot}\pi_1\ \rho_1\ \iota_1 = C\,[\![\texttt{e1}]\!]\ \$\bot\ \rho\ \iota;$
$[\![\texttt{e2}_2]\!]\ \alpha_2{\cdot}\pi_2\ \rho_2\ \iota_2 = C\,[\![\texttt{e2}]\!]\ \alpha{\cdot}\pi\ \rho_1\ \iota;$
$[\![\texttt{e3}_3]\!]\ \alpha_3{\cdot}\pi_3\ \rho_3\ \iota_3 = C\,[\![\texttt{e3}]\!]\ \alpha{\cdot}\pi\ \rho_1\ \iota;$

$$\rho_4 = \lambda i.\begin{cases}\langle [\![\texttt{binding}_2]\!],\ (pa_2 \sqcap pa_3)\ , 0, b\text{-}type_2\rangle \\ \qquad \text{if } b\text{-}type_2 = \textbf{lambda}; \\ \langle [\![\texttt{binding}_2]\!], pa_4, v\text{-}count_2 + v\text{-}count_3, b\text{-}type_2\rangle \\ \qquad \text{if } b\text{-}type_2 = \textbf{fix}\ ;\end{cases}$$

**where**

$\langle [\![\texttt{binding}_2]\!], pa_2, v\text{-}count_2, b\text{-}type_2\rangle = \rho_2\ i$
$\langle [\![\texttt{binding}_3]\!], pa_3, v\text{-}count_3, b\text{-}type_3\rangle = \rho_3\ i$

$$pa_4 = \lambda pat.\begin{cases}(pa_3\ pat) & \text{if } (pa_2\ pat) = unbound; \\ (pa_2\ pat) & \text{otherwise.}\end{cases}$$

**Case 7:** $[\![\texttt{exp}]\!] = [\![\texttt{if:<e1 e2 e3>}]\!]$

**Given:** $\alpha_d{\cdot}\pi_d, R, R'$ *such that*

i) $\bot_S \notin I\,[\![\texttt{if:<e1 e2 e3>}]\!]\ \alpha_d{\cdot}\pi_d\ R$

ii) *Safe-pat* $(\alpha{\cdot}\pi, \alpha_d{\cdot}\pi_d)$

iii) *Safe-comp-env* $(\rho)$

iv) *Equivalently-extended* $(\rho, R, R')$

**Need to show:**

$(I\,[\![\texttt{if:<e1 e2 e3>}]\!]\ \alpha_d{\cdot}\pi_d\ R = I\,[\![\texttt{if:<\$e1}_1\ \texttt{e2}_2\ \texttt{e3}_3\texttt{>}]\!]\ \alpha_d{\cdot}\pi_d\ R')$ &
*Safe-comp-env* $(\rho_4)$

**Proof:** As shown in Case 3), there are three cases.

Case 7.1) $(\$ \notin \alpha_d{\cdot}\pi_d)$ & $(\$ \notin \alpha{\cdot}\pi)$; See Case 2).

Case 7.2) $(\$ \in \alpha_d{\cdot}\pi_d)$ & $(\$ \notin \alpha{\cdot}\pi)$; See Case 2).

Case 7.3) $(\$ \in \alpha_d{\cdot}\pi_d)$ & $(\$ \in \alpha{\cdot}\pi)$

Steps (1) and (2) justify the use of the induction hypothesis in step (3). Steps (4) and (5) allow us to then show that the compiler's object code will satisfy the invariant for all possible values returned by the interpretation of the predicate, in steps (6) and (7). Steps (10) through (15) show that the environment invariant, *Safe-comp-env* ($\rho_4$, $[\![\texttt{ide}]\!]$ ) is maintained.

$\perp_S \notin I[\![\texttt{e1}]\!]\ \$\perp\ R$ (1)

By i) and a7).

*Safe-pat* ($\$\perp, \$\perp$) (2)

$(I[\![\texttt{e1}]\!]\ \$\perp\ R\ = I[\![\texttt{e1}_1]\!]\ \$\perp\ R')$
& *Safe-comp-env* ($\rho_1$) (3)

By (1), (2), iii) and IH.

$\perp_S \notin I[\![\texttt{e2}]\!]\ \alpha_d{\cdot}\pi_d\ R \implies$
$\Big((I[\![\texttt{e2}]\!]\ \alpha_d{\cdot}\pi_d\ R\ = I[\![\texttt{e2}_2]\!]\ \alpha_d{\cdot}\pi_d\ R')$ & *Safe-comp-env* $(\rho_2)\Big)$ (4)

By ii), (3) and IH.

$\perp_S \notin I[\![\texttt{e3}]\!]\ \alpha_d{\cdot}\pi_d\ R \implies$
$\Big((I[\![\texttt{e3}]\!]\ \alpha_d{\cdot}\pi_d\ R\ = I[\![\texttt{e3}_3]\!]\ \alpha_d{\cdot}\pi_d\ R')$ & *Safe-comp-env* $(\rho_3)\Big)$ (5)

By ii), (3) and IH.

$I[\![\texttt{e1}]\!]\ \$\perp\ R \neq \mathit{false} \implies (I[\![\texttt{e2}]\!]\ \alpha_d{\cdot}\pi_d\ R\ = I[\![\texttt{e2}_2]\!]\ \alpha_d{\cdot}\pi_d\ R')$ (6)

By i), a7) and (4).

$I[\![\texttt{e1}]\!]\ \$\perp\ R\ = \mathit{false} \implies (I[\![\texttt{e3}]\!]\ \alpha_d{\cdot}\pi_d\ R\ = I[\![\texttt{e3}_3]\!]\ \alpha_d{\cdot}\pi_d\ R')$ (7)

By i), a7) and (5).

$I[\![\texttt{if:<e1 e2 e3>}]\!]\ \alpha_d{\cdot}\pi_d\ R\ = I[\![\texttt{if:<\$e1}_1\ \texttt{e2}_2\ \texttt{e3}_3\texttt{>}]\!]\ \alpha_d{\cdot}\pi_d\ R'$ (8)

By (3), (6), (7), a1) and substitution.

$b\text{-}type \in \{\mathbf{lambda}, \mathbf{fix}\}$ (9)

By a7).

$b\text{-}type = \mathbf{lambda}$ (10)

Given.

$(pa_2 \sqcap pa_3) \sqsubseteq pa_2 \;\; \& \;\; (pa_2 \sqcap pa_3) \sqsubseteq pa_3$ (11)

By ($\mathcal{C}$ 7).

$\forall$ $[\![\texttt{ide}]\!]$

$(\rho_4 \; [\![\texttt{ide}]\!]) \neq unbound \implies$

$((Binding\text{-}type \; (\rho_4 \; [\![\texttt{ide}]\!])) = \mathbf{lambda}) \implies Safe\text{-}lambda\text{-}exp \; (\rho_4, [\![\texttt{ide}]\!])$ (12)

By (4), (5),(10) and (11).

$b\text{-}type = \mathbf{fix}$ (13)

Given.

$\forall \; \alpha{\cdot}\pi_{inh} \in dom \; pa_4$

$\alpha{\cdot}\pi_{inh} \in dom \; pa_2 \;\; \vee \;\; \alpha{\cdot}\pi_{inh} \in dom \; pa_3$ (14)

By ($\mathcal{C}$ 7).

$\forall$ $[\![\texttt{ide}]\!]$

$(\rho_4 \; [\![\texttt{ide}]\!]) \neq unbound \implies$

$((Binding\text{-}type \; (\rho_4 \; [\![\texttt{ide}]\!])) = \mathbf{fix}) \implies Safe\text{-}fix\text{-}exp \; (\rho_4, [\![\texttt{ide}]\!])$ (15)

By (4), (5), (13) and (14).

$Safe\text{-}comp\text{-}env \; (\rho_4)$

By (12) and (15). □

$$C\,[\![(\lambda \mathtt{id.\ body}):\mathtt{e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota = \qquad (\mathcal{C}\ 8)$$
$$[\![(\lambda \mathtt{id.\ body_1}):\mathtt{e_1}]\!]\ \alpha{\cdot}\pi\ \rho_4\ \iota$$

**where**

$$[\![\mathtt{body_1}]\!]\ \alpha_1{\cdot}\pi_1\ \rho_1\ \iota_1 = C\,[\![\mathtt{body}]\!]\ \alpha{\cdot}\pi\ \rho_2\ \iota$$

$$\rho_2 = \lambda i.\begin{cases}\langle [\![\,[]\,]\!], \bot, 0, \mathbf{lambda}\rangle, & \text{if } i = [\![\mathtt{id}]\!];\\ \rho\, i, & \text{otherwise;}\end{cases}$$

$$\rho_3 = \lambda i.\begin{cases}\rho\, i, & \text{if } i = [\![\mathtt{id}]\!];\\ \rho_1\, i, & \text{otherwise;}\end{cases}$$

$$[\![\mathtt{e_1}]\!]\ \alpha_2{\cdot}\pi_2\ \rho_4\ \iota_2 = C\,[\![\mathtt{e}]\!]\ (\mathit{Pat\text{-}fun}\ (\rho_1\ [\![\mathtt{id}]\!]))\ \rho_3\ \iota.$$

**Case 8:** $[\![\mathtt{exp}]\!] = [\![(\lambda \mathtt{id.\ body}):\mathtt{e}]\!]$

**Given:** $\alpha_d{\cdot}\pi_d, R, R'$ *such that*

i) $\bot_S \notin I\,[\![(\lambda \mathtt{id.\ body}):\mathtt{e}]\!]\ \alpha_d{\cdot}\pi_d\ R$

ii) *Safe-pat* $(\alpha{\cdot}\pi, \alpha_d{\cdot}\pi_d)$

iii) *Safe-comp-env* $(\rho)$

iv) *Equivalently-extended* $(\rho, R, R')$

**Need to show:**

$(I\,[\![(\lambda \mathtt{id.\ body}):\mathtt{e}]\!]\ \alpha_d{\cdot}\pi_d\ R = I\,[\![(\lambda \mathtt{id.\ body_1}):\mathtt{e_1}]\!]\ \alpha_d{\cdot}\pi_d\ R')$

&

*Safe-comp-env* $(\rho_4)$

**Proof:**

As shown in Case 3), there are three cases.

Case 8.1) $(\$ \notin \alpha_d{\cdot}\pi_d)$ & $(\$ \notin \alpha{\cdot}\pi)$; See Case 2).

Case 8.2) $(\$ \in \alpha_d{\cdot}\pi_d)$ & $(\$ \notin \alpha{\cdot}\pi)$; See Case 2).

Case 8.3) $(\$ \in \alpha_d{\cdot}\pi_d)$ & $(\$ \in \alpha{\cdot}\pi)$

Steps (1) through (7), following, justify the use of the induction hypothesis which shows that the compilation of the lambda body is safe. Steps (2) through (7) show that the environment given to the compilation of the lambda body is

safe. Steps (8) through (10) justify the use of the induction hypothesis, showing that the compilation of the argument $[\![\texttt{e}]\!]$ is safe.

$\perp_S \notin I[\![\texttt{body}]\!]\ \alpha_d{\cdot}\pi_d\ R[\,[\![\texttt{e}]\!]\,/\,[\![\texttt{id}]\!]\,]$ (1)

By i) and a8)

$(\rho_2\ [\![\texttt{id}]\!]) \neq \mathit{unbound}$ (2)

By ($\mathcal{C}$ 8).

$\mathit{b\text{-}type} = \textbf{lambda}$ (3)

Given.

$\forall\ \alpha{\cdot}\pi \in P,\ \perp \sqsubseteq \alpha{\cdot}\pi$ (4)

By defn. of $\perp$.

$\mathit{Safe\text{-}lambda\text{-}exp}\ (\rho_2,\ [\![\texttt{id}]\!])$ (5)

By (2), (3), (4) and iii).

$\forall\ [\![\texttt{ide}]\!]$
$(\rho_2\ [\![\texttt{ide}]\!]) \neq \mathit{unbound} \implies$
$((\mathit{Binding\text{-}type}\ (\rho_2\ [\![\texttt{ide}]\!])) = \textbf{lambda}) \implies \mathit{Safe\text{-}lambda\text{-}exp}\ (\rho_2,\ [\![\texttt{ide}]\!])$ (6)

By iii) and (5).

$\mathit{Safe\text{-}comp\text{-}env}\ (\rho_2)$ (7)

By iii) and (6).

$(I[\![\texttt{body}]\!]\ \alpha_d{\cdot}\pi_d\ R[\,[\![\texttt{e}]\!]\,/\,[\![\texttt{id}]\!]\,] = I[\![\texttt{body}_1]\!]\ \alpha_d{\cdot}\pi_d\ R'[\,[\![\texttt{e}]\!]\,/\,[\![\texttt{id}]\!]\,])$
&
$\mathit{Safe\text{-}comp\text{-}env}\ (\rho_1)$ (8)

By (1), ii), (7) and IH.

$\perp_S \notin I[\![\texttt{e}]\!]\ (\sqcup_{i=1}^{n}([\![\texttt{id}]\!]_i \blacksquare I[\![(\lambda\texttt{id.body}):\texttt{e}]\!]\ \alpha_d{\cdot}\pi_d\ R))\ R$ (9)

By i) and a8).

$\mathit{Safe\text{-}comp\text{-}env}\ (\rho_3)$ (10)

By iii) and (8).

$(I[\![\texttt{e}]\!]\ (\sqcup_{i=1}^{n}([\![\texttt{id}]\!]_i \blacksquare I[\![(\lambda\texttt{id. body}):\texttt{e}]\!]\ \alpha_d{\cdot}\pi_d\ R))\ R =$
$I[\![\texttt{e}_1]\!]\ (\sqcup_{i=1}^{n}([\![\texttt{id}]\!]_i \blacksquare I[\![(\lambda\texttt{id. body}):\texttt{e}_1]\!]\ \alpha_d{\cdot}\pi_d\ R'))\ R')$
& $\mathit{Safe\text{-}comp\text{-}env}\ (\rho_4)$ (11)

By (9), (8), (10) and IH.

$$I[\![(\lambda \texttt{id. body}) : \texttt{e}]\!]\ \alpha_d \cdot \pi_d\ R = I[\![(\lambda \texttt{id. body}_1) : \texttt{e}_1]\!]\ \alpha_d \cdot \pi_d\ R'$$

By (8), (10) and substitution. □

$$
\begin{array}{l}
C\,[\![\texttt{(fix:[f }\lambda\texttt{id. body])}:\texttt{e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota = \\
\qquad [\![\texttt{(fix:[f}|\alpha{\cdot}\pi\ \lambda\texttt{id. body}_1\texttt{])}:\texttt{e}_1]\!]\ \alpha{\cdot}\pi\ \rho_4\ \iota \\
\textbf{where} \\
[\![\texttt{body}_1]\!]\ \alpha_1{\cdot}\pi_1\ \rho_1\ \iota_1 = C\,[\![\texttt{body}]\!]\ \alpha{\cdot}\pi\ \rho_2\ \iota \\
\rho_2 = \lambda i.\begin{cases} \langle\, [\![\texttt{(fix:[f }\lambda\texttt{id. body])}]\!], pa, 1, \textbf{fix}\,\rangle, & \text{if } i = [\![\texttt{f}]\!]; \\ \langle\, [\![\texttt{[]}]\!], \bot, 0, \textbf{lambda}\rangle, & \text{if } i = [\![\texttt{id}]\!]; \\ \rho\, i, & \text{otherwise;} \end{cases} \\
pa = \lambda pat.\begin{cases} \textit{rec-p} & \text{if } pat = \alpha{\cdot}\pi; \\ \textit{unbound} & \text{otherwise;} \end{cases} \\
\rho_3 = \lambda i.\begin{cases} \rho\, i, & \text{if } i = [\![\texttt{id}]\!] \textit{ or } i = [\![\texttt{f}]\!]; \\ \rho_1\, i, & \text{otherwise;} \end{cases} \\
\textit{rec-p} = [\![\texttt{e}]\!] \bullet C\,[\![(\sqcup_{u=1}^{\infty}(U_u\ \texttt{(fix:[f }\lambda\texttt{id. body])}))\texttt{:e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota \\
\qquad = (\rho_1\ [\![\texttt{id}]\!])\downarrow 2\downarrow 1 \\
[\![\texttt{e}_1]\!]\ \alpha_2{\cdot}\pi_2\ \rho_4\ \iota_2 = C\,[\![\texttt{e}]\!]\ \textit{rec-p}\ \rho_3\ \iota.
\end{array}
\qquad (\mathcal{C}\,9)
$$

**Case 9:** $[\![\texttt{exp}]\!] = [\![\texttt{(fix:[f }\lambda\texttt{id. body])}:\texttt{e}]\!]$

**Given:** $\alpha_d{\cdot}\pi_d, R, R'$ *such that*

i) $\bot_S \notin I\,[\![\texttt{(fix:[f }\lambda\texttt{id. body])}:\texttt{e}]\!]\ \alpha_d{\cdot}\pi_d\ R$

ii) *Safe-pat* $(\alpha{\cdot}\pi, \alpha_d{\cdot}\pi_d)$

iii) *Safe-comp-env* $(\rho)$

iv) *Equivalently-extended* $(\rho, R, R')$

**Need to show:**

$(I\,[\![\texttt{(fix:[f }\lambda\texttt{id. body])}:\texttt{e}]\!]\ \alpha_d{\cdot}\pi_d\ R =$

$\quad I\,[\![\texttt{(fix:[f}|\alpha{\cdot}\pi\ \lambda\texttt{id. body}_1\texttt{])}:\texttt{e}_1]\!]\ \alpha_d{\cdot}\pi_d\ R')$

&

*Safe-comp-env* $(\rho_4)$

The case showing that $C$ safely compiles recursive functions depends upon first proving that all finite unfoldings of a recursive function definition are compiled safely. This makes use of one of the other cases of the induction, the proof that $\lambda$-expressions are safely handled. However, the number of such unfoldings

needed at compile-time to produce the final result at run-time is unbounded, and may be infinite if the result is an infinite list. For this reason, the compiler is additionally proven safe when it receives the least upper bound of these approximations, the unfolded equivalent to the finite *fix* expression actually compiled. This least upper bound only exists in a special domain of syntactic expressions described and used only in this chapter; it is not specified by the grammar in Chapter 2, which only constructs finite expressions. In Section 3.2, $C$ is shown to be monotonic and continuous using this domain, which then makes it possible to construct a lemma showing that $C$ produces a synthesized pattern no higher than that displayed by $I$ when $I$ evaluates an application of a recursive function.

**Lemma 3.2.1:** $\textit{rec-p} \sqsubseteq [\![\texttt{e}]\!] \blacksquare I [\![(\sqcup_{u=1}^{\infty} (U_u\ \texttt{(fix:[f λid. body])})) : \texttt{e}]\!]\ \alpha_d \cdot \pi_d\ R$

**Proof:**

$\textit{rec-p} = [\![\texttt{e}]\!] \bullet C [\![(\sqcup_{u=1}^{\infty} (U_u\ \texttt{(fix:[f λid. body])})) : \texttt{e}]\!]\ \alpha \cdot \pi\ \rho\ \iota$

Case 8 shows that for any finite unfolding $u$ of $[\![\texttt{(fix:[f λid. body])}]\!]$, which becomes a simple lambda expression,

$[\![\texttt{e}]\!] \bullet C [\![(U_u\ \texttt{(fix:[f λid. body])}) : \texttt{e}]\!]\ \alpha \cdot \pi\ \rho\ \iota$

$\sqsubseteq$

$[\![\texttt{e}]\!] \blacksquare I [\![(U_u\ \texttt{(fix:[f λid. body])}) : \texttt{e}]\!]\ \alpha_d \cdot \pi_d\ R.$

It is shown that for any finite unfolding $u$,

$[\![\texttt{e}]\!] \bullet C [\![(U_u\ \texttt{(fix:[f λid. body])}) : \texttt{e}]\!]\ \alpha \cdot \pi\ \rho\ \iota$

$\sqsubseteq$

$[\![\texttt{e}]\!] \bullet C [\![(U_{u+1}\ \texttt{(fix:[f λid. body])}) : \texttt{e}]\!]\ \alpha \cdot \pi\ \rho\ \iota$

by observing that the patterns inherited by the compilation of the extra unfolding are at least $\perp$, thus preserving the patterns inherited by the compilation of $u$ unfoldings. A similar observation shows that

$[\![\texttt{e}]\!] \blacksquare I [\![(U_u\ \texttt{(fix:[f λid. body])}) : \texttt{e}]\!]\ \alpha_d \cdot \pi_d\ R$

$\sqsubseteq$

$[\![\texttt{e}]\!] \blacksquare I [\![(U_{u+1}\ \texttt{(fix:[f λid. body])}) : \texttt{e}]\!]\ \alpha_d \cdot \pi_d\ R.$

By Theorem 3.1), $C$ is a continuous functional, and so

$C\,[\![(\sqcup_{u=1}^{\infty}(U_u\ \texttt{(fix:[f }\lambda\texttt{id. body])})):\texttt{e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota$

$=$

$\sqcup_{u=1}^{\infty}\ (C\,[\![(U_u\ \texttt{(fix:[f }\lambda\texttt{id. body])):e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota)$

From this,

$C\,[\![\texttt{body}_1]\!]\ \alpha{\cdot}\pi\ \rho'\ \iota$

$=$

$\sqcup_{u=1}^{\infty}\ (C\,[\![\texttt{body}_2]\!]\ \alpha{\cdot}\pi\ \rho'\ \iota)$

**where** $[\![\lambda\texttt{id. body}_1]\!] = [\![(\sqcup_{u=1}^{\infty}(U_u\ \texttt{(fix:[f }\lambda\texttt{id. body])}))]\!]$

$[\![\lambda\texttt{id. body}_2]\!] = [\![(U_u\ \texttt{(fix:[f }\lambda\texttt{id. body])})]\!]$

and

$Pat\text{-}fun\ (\rho_1\ [\![\texttt{id}]\!]) = Pat\text{-}fun\ (\rho_2\ [\![\texttt{id}]\!])$

**where** $\rho_1 = (C\,[\![\texttt{body}_1]\!]\ \alpha{\cdot}\pi\ \rho'\ \iota)\downarrow 2\downarrow 2\downarrow 1$

$\rho_2 = (\sqcup_{u=1}^{\infty}\ (C\,[\![\texttt{body}_2]\!]\ \alpha{\cdot}\pi\ \rho'\ \iota))\downarrow 2\downarrow 2\downarrow 1$

$[\![\lambda\texttt{id. body}_1]\!] = [\![(\sqcup_{u=1}^{\infty}(U_u\ \texttt{(fix:[f }\lambda\texttt{id. body])}))]\!]$

$[\![\lambda\texttt{id. body}_2]\!] = [\![(U_u\ \texttt{(fix:[f }\lambda\texttt{id. body])})]\!]$

By ($C$ 9),

$Pat\text{-}fun\ (\rho_1\ [\![\texttt{id}]\!]) =$

$[\![\texttt{e}]\!] \bullet C\,[\![(\sqcup_{u=1}^{\infty}(U_u\ \texttt{(fix:[f }\lambda\texttt{id. body])})):\texttt{e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota)$

and

$Pat\text{-}fun\ (\rho_2\ [\![\texttt{id}]\!]) =$

$\sqcup_{u=1}^{\infty}\ ([\![\texttt{e}]\!] \bullet C\,[\![(U_u\ \texttt{(fix:[f }\lambda\texttt{id. body])):e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota)$

Thus,

$[\![\texttt{e}]\!] \bullet C\,[\![(\sqcup_{u=1}^{\infty}(U_u\ \texttt{(fix:[f }\lambda\texttt{id. body])})):\texttt{e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota)$

$=$

$\sqcup_{u=1}^{\infty}\ ([\![\texttt{e}]\!] \bullet C\,[\![(U_u\ \texttt{(fix:[f }\lambda\texttt{id. body])):e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota)$

$[\![\texttt{e}]\!] \blacksquare I\,[\![(\sqcup_{u=1}^{\infty}(U_u\ \texttt{(fix:[f }\lambda\texttt{id. body])})):\texttt{e}]\!]\ \alpha_d{\cdot}\pi_d\ R$

forms an upper bound for the patterns displayed during the interpretation of all finite approximations (unfoldings) of $[\![(\texttt{fix:[f } \lambda\texttt{id. body]})]\!]$, so it must be at least as high as $[\![\texttt{e}]\!] \bullet C\,[\![(\sqcup_{u=1}^{\infty}(U_u\ (\texttt{fix:[f } \lambda\texttt{id. body]}))):\texttt{e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota$ since this is their *least* upper bound. Thus,

$[\![\texttt{e}]\!] \bullet C\,[\![(\sqcup_{u=1}^{\infty}(U_u\ (\texttt{fix:[f } \lambda\texttt{id. body]}))):\texttt{e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota$
$\sqsubseteq$
$[\![\texttt{e}]\!]\ {\blacksquare} I\,[\![(\sqcup_{u=1}^{\infty}(U_u\ (\texttt{fix:[f } \lambda\texttt{id. body]}))):\texttt{e}]\!]\ \alpha_d{\cdot}\pi_d\ R$ □

**Proof:**

As shown in Case 3), there are three possible cases.

Case 9.1) ($\$ \notin \alpha_d{\cdot}\pi_d$) & ($\$ \notin \alpha{\cdot}\pi$); See Case 2).

Case 9.2) ($\$ \in \alpha_d{\cdot}\pi_d$) & ($\$ \notin \alpha{\cdot}\pi$); See Case 2).

Case 9.3) ($\$ \in \alpha_d{\cdot}\pi_d$) & ($\$ \in \alpha{\cdot}\pi$)

Steps (1) through (11) justify the use of the induction hypothesis in showing that the body of the lambda expression is compiled safely. Steps (2) through (11) show that the environment given to the compilation of the lambda body is safe. Steps (13) and (14) justify the use of the induction hypothesis in showing that the compilation of the application argument $[\![\texttt{e}]\!]$ is safe.

$\perp_S \notin I\,[\![\texttt{body}]\!]\ \alpha_d{\cdot}\pi_d\ R[\,[\![(\texttt{fix:[f } \lambda\texttt{id. body]})]\!] / [\![\texttt{f}]\!]\,]$ (1)

By i) and a9).

$(\rho_2\ [\![\texttt{f}]\!]) \neq \mathit{unbound}$ (2)

By ($\mathcal{C}$ 9).

$(\mathit{Binding\text{-}type}\ (\rho_2\ [\![\texttt{f}]\!])) = \mathbf{fix}$ (3)

Given.

$\mathit{Safe\text{-}fix\text{-}exp}\ (\rho_2, [\![\texttt{f}]\!])$ (4)

By (2), (3), ii) and Lemma 3.2.1.

$\forall\ [\![\texttt{ide}]\!]$
$(\rho_2\ [\![\texttt{ide}]\!]) \neq \mathit{unbound} \implies$
$(\mathit{Binding\text{-}type}\ (\rho_2\ [\![\texttt{ide}]\!])) = \mathbf{fix} \implies \mathit{Safe\text{-}fix\text{-}exp}\ (\rho_2, [\![\texttt{ide}]\!])$ (5)

By (4) and iii).

(6)

$(\rho_2\ [\![\texttt{id}]\!]) \neq \mathit{unbound}$ By ($\mathcal{C}$ 9).

(7)

$(\mathit{Binding\text{-}type}\ (\rho_2\ [\![\texttt{id}]\!])) = \mathbf{lambda}$ Given.

(8)

$\forall\ \alpha{\cdot}\pi \in P,\ \bot \sqsubseteq \alpha{\cdot}\pi$ By defn. of $\bot$.

(9)

$\mathit{Safe\text{-}lambda\text{-}exp}\ (\rho_2\ [\![\texttt{id}]\!])$ By (6), (7) and (8).

$\forall\ [\![\texttt{ide}]\!]$
$(\rho_2\ [\![\texttt{ide}]\!]) \neq \mathit{unbound} \implies$
$(\mathit{Binding\text{-}type}\ (\rho_2\ [\![\texttt{ide}]\!])) = \mathbf{lambda}) \implies \mathit{Safe\text{-}lambda\text{-}exp}\ (\rho_2, [\![\texttt{ide}]\!])$ (10)
By (9) and iii).

(11)

$\mathit{Safe\text{-}comp\text{-}env}\ (\rho_2)$ By (5) and (10).

$(I[\![\texttt{body}]\!]\ \alpha_d{\cdot}\pi_d\ R[\ [\![\texttt{(fix:[f }\lambda\texttt{id. body]):e}]\!] / [\![\texttt{f}]\!]\ ] =$
$I[\![\texttt{body}_1]\!]\ \alpha_d{\cdot}\pi_d\ R'[\ [\![\texttt{(fix:[f}|\alpha{\cdot}\pi\ \lambda\texttt{id. body}_1\texttt{]):e}]\!] / [\![\texttt{f}|\alpha{\cdot}\pi]\!]\ ])$
&
(12)

$\mathit{Safe\text{-}comp\text{-}env}\ (\rho_1)$ By (1), ii), (11) and IH.

(13)

$\bot_S \notin I[\![\texttt{e}]\!]\ [\![\texttt{e}]\!]\ \blacksquare I[\![(\sqcup_{u=1}^{\infty}(U_u\ \texttt{(fix:[f }\lambda\texttt{id. body]))):e}]\!]\ \alpha_d{\cdot}\pi_d\ R)\ R$ By i) and a9)

(14)

$\mathit{Safe\text{-}comp\text{-}env}\ (\rho_3)$ By iii) and (12).

$I[\![\texttt{e}]\!]\ (\ [\![\texttt{e}]\!]\ \blacksquare I[\![(\sqcup_{u=1}^{\infty}\ (U_u\ \texttt{(fix:[f }\lambda\texttt{id. body]))):e}]\!]\ \alpha_d{\cdot}\pi_d\ R)\ R\ =$
$I[\![\texttt{e}_1]\!]\ (\ [\![\texttt{e}]\!]\ \blacksquare I[\![(\sqcup_{u=1}^{\infty}\ (U_u\ \texttt{(fix:[f}|\alpha{\cdot}\pi\ \lambda\texttt{id. body}_1\texttt{]))):e}_1]\!]\ \alpha_d{\cdot}\pi_d\ R')\ R'$
&
(15)

$\mathit{Safe\text{-}comp\text{-}env}\ (\rho_4)$ By (13), Lemma 3.2.1), (14) and IH.

$I[\![\texttt{(fix:[f }\lambda\texttt{id. body]):e}]\!]\ \alpha_d{\cdot}\pi_d\ R\ =$

$I[\![\texttt{(fix:[f}|\alpha\cdot\pi\ \lambda\texttt{id. body}_1\texttt{])}:\texttt{e}_1]\!]\ \alpha_d\cdot\pi_d\ R'$

By (12), (15) and substitution. □

$$C\,[\![\texttt{fix:[id e]}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota = [\![\texttt{fix:[id}|\alpha{\cdot}\pi\ \texttt{e}_1\texttt{]}]\!]\ \alpha{\cdot}\pi\ \rho_3\ \iota_1 \qquad (C\ 10)$$

**where**

$[\![\texttt{e}_1]\!]\ \alpha_1{\cdot}\pi_1\ \rho_1\ \iota_1 = C\,[\![\texttt{e}]\!]\ \alpha{\cdot}\pi\ \rho_2\ \iota;$

$$\rho_2 = \lambda i.\begin{cases} \langle\, [\![\texttt{fix:[id e]}]\!], pa, 1, \textbf{fix}\,\rangle, \\ \quad \text{if } i = [\![\texttt{id}]\!]; \\ \rho\, i, \text{otherwise;} \end{cases}$$

$$pa = \lambda pat.\begin{cases} \bot & \text{if } pat = \alpha{\cdot}\pi; \\ unbound & \text{otherwise;} \end{cases}$$

$$\rho_3 = \lambda i.\begin{cases} \rho\, i, & \text{if } i = [\![\texttt{id}]\!]; \\ \rho_1\, i, & \text{otherwise.} \end{cases}$$

**Case 10:** $[\![\texttt{exp}]\!] = [\![\texttt{fix:[id e]}]\!]$

**Given:** $\alpha_d{\cdot}\pi_d, R, R'$ *such that*

i) $\bot_S \notin I\,[\![\texttt{fix:[id e]}]\!]\ \alpha_d{\cdot}\pi_d\ R$

ii) *Safe-pat* $(\alpha{\cdot}\pi, \alpha_d{\cdot}\pi_d)$

iii) *Safe-comp-env* $(\rho)$

iv) *Equivalently-extended* $(\rho, R, R')$

**Need to show:**

$(I\,[\![\texttt{fix:[id e]}]\!]\ \alpha_d{\cdot}\pi_d\ R = I\,[\![\texttt{fix:[id}|\alpha{\cdot}\pi\ \texttt{e}_1\texttt{]}]\!]\ \alpha_d{\cdot}\pi_d\ R')$

& *Safe-comp-env* $(\rho_3)$

As shown in Case 3), there are three possible cases.

Case 10.1) $(\$ \notin \alpha_d{\cdot}\pi_d)$ & $(\$ \notin \alpha{\cdot}\pi)$; See Case 2).

Case 10.2) $(\$ \in \alpha_d{\cdot}\pi_d)$ & $(\$ \notin \alpha{\cdot}\pi)$; See Case 2).

Case 10.3) $(\$ \in \alpha_d{\cdot}\pi_d)$ & $(\$ \in \alpha{\cdot}\pi)$

Steps (1) through (7) justify the use of the induction hypothesis in showing that the compilation of $[\![\texttt{e}]\!]$ is safe. Steps (2) though (7) show that the environment given to the compilation of $[\![\texttt{e}]\!]$.

(1)

$\perp_S \notin I[\![\mathtt{e}]\!]\ \alpha_d{\cdot}\pi_d\ R[\,[\![\mathtt{e}]\!]\,/\,[\![\mathtt{id}]\!]\,]$ By i) and a10).

(2)

$(\rho_2\ [\![\mathtt{id}]\!]) \neq \mathit{unbound}$ By ($\mathcal{C}$ 10).

(3)

$(\mathit{Binding\text{-}type}\ (\rho_2\ [\![\mathtt{id}]\!])) = \mathbf{fix}$ Given.

(4)

$\forall\ \alpha{\cdot}\pi \in P, \perp \sqsubseteq \alpha{\cdot}\pi$ By defn. of $\perp$.

(5)

$\mathit{Safe\text{-}fix\text{-}exp}\ (\rho_2\ [\![\mathtt{id}]\!])$ By (4) and ii).

$\forall\ [\![\mathtt{ide}]\!]$

(6)

$(\rho_2\ [\![\mathtt{ide}]\!]) \neq \mathit{unbound} \implies$
$(\mathit{Binding\text{-}type}\ (\rho_2\ [\![\mathtt{ide}]\!])) = \mathbf{fix} \implies \mathit{Safe\text{-}fix\text{-}exp}\ (\rho_2\ [\![\mathtt{ide}]\!])$
By (2), (3), (5), and iii).

(7)

$\mathit{Safe\text{-}comp\text{-}env}\ (\rho_2)$ By (6) and iii).

(8)

$(I[\![\mathtt{e}]\!]\ \alpha_d{\cdot}\pi_d\ R[\,[\![\mathtt{fix:[id\ e]}]\!]\,/\,[\![\mathtt{id}]\!]\,] =$
$I[\![\mathtt{e}_1]\!]\ \alpha_d{\cdot}\pi_d\ R'[\,[\![\mathtt{fix:[id}|\alpha{\cdot}\pi\ \mathtt{e}_1\mathtt{]}]\!]\,/\,[\![\mathtt{id}|\alpha{\cdot}\pi]\!]\,])$
$\&\ \mathit{Safe\text{-}comp\text{-}env}\ (\rho_1)$
By (1), ii), (7) and IH.

(9)

$I[\![\mathtt{fix:[id\ e]}]\!]\ \alpha_d{\cdot}\pi_d\ R = I[\![\mathtt{fix:[id}|\alpha{\cdot}\pi\ \mathtt{e}_1\mathtt{]}]\!]\ \alpha_d{\cdot}\pi_d\ R'$
By (8) and substitution.

$\mathit{Safe\text{-}comp\text{-}env}\ (\rho_3)$ By iii) and (8). □

(C 11)

$C\,[\![\texttt{f:e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota =$

$$\begin{cases} \textbf{Reached-Limit} & \text{if } (pa\ \alpha{\cdot}\pi) = \textit{unbound}\ \&\ \textit{v-count} \geq \iota; \\ \textbf{Compile-Binding} & \text{if } (pa\ \alpha{\cdot}\pi) = \textit{unbound}\ \&\ \textit{v-count} < \iota; \\ \textbf{Mark-With-Pattern} & \text{otherwise;} \end{cases}$$

**where** $\langle\, [\![\texttt{(fix:[f }\lambda\texttt{id. body])}]\!]\,, pa, \textit{v-count}, \textbf{fix}\,\rangle = \rho\ [\![\texttt{f}]\!]\,;$

**Reached-Limit is** $[\![\texttt{(fix:[f }\lambda\texttt{id.body]):e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota;$

**Compile-Binding is**
$[\![\texttt{(fix:[f}|\alpha{\cdot}\pi\ \lambda\texttt{id. body}_1\texttt{])}:\texttt{e}_1]\!]\ \alpha{\cdot}\pi\ \rho_4\ \iota$
**where**

$$[\![\texttt{body}_1]\!]\ \alpha_1{\cdot}\pi_1\ \rho_1\ \iota_1 = C\,[\![\texttt{body}]\!]\ \alpha{\cdot}\pi\ \rho_2\ \iota, \quad \text{if } i = [\![\texttt{f}]\!]\,;$$

$$\rho_2 = \lambda i. \begin{cases} \langle\, [\![\texttt{(fix:[f }\lambda\texttt{id. body])}]\!]\,, pa_1, \textit{v-count}+1, \textbf{fix}\,\rangle, & \text{if } i = [\![\texttt{f}]\!]\,; \\ \langle\, [\![\texttt{[]}]\!]\,, \bot, 0, \textbf{lambda}\rangle, & \text{if } i = [\![\texttt{id}]\!]\,; \\ \rho\, i, & \text{otherwise;} \end{cases}$$

$$pa_1 = \lambda pat. \begin{cases} \textit{rec-p} & \text{if } pat = \alpha{\cdot}\pi; \\ pa & \text{otherwise;} \end{cases}$$

$$\rho_3 = \lambda i. \begin{cases} \rho\, i, & \text{if } i = [\![\texttt{id}]\!] \textit{ or } i = [\![\texttt{f}]\!]\,; \\ \rho_1\, i, & \text{otherwise;} \end{cases}$$

$$\textit{rec-p} = [\![\texttt{e}]\!] \bullet C\,[\![(\sqcup_{u=1}^{\infty}(U_u\ \texttt{(fix:[f }\lambda\texttt{id. body])}))\texttt{:e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota$$
$$= (\rho_1\ [\![\texttt{id}]\!]\,)\downarrow 2\downarrow 1$$
$$[\![\texttt{e}_1]\!]\ \alpha_2{\cdot}\pi_2\ \rho_4\ \iota_2 = C\,[\![\texttt{e}]\!]\ \textit{rec-p}\ \rho_3\ \iota$$

**Mark-With-Pattern is**
$[\![\texttt{f}|\alpha{\cdot}\pi\texttt{:e}_1]\!]\ \alpha{\cdot}\pi\ \rho_1\ \iota$
**where**
$[\![\texttt{e}_1]\!]\ \alpha_1{\cdot}\pi_1\ \rho_1\ \iota_1 = C\,[\![\texttt{e}]\!]\ (pa\ \alpha{\cdot}\pi)\ \rho\ \iota$

**Case 11:** $[\![\texttt{exp}]\!] = [\![\texttt{f:e}]\!]$

**Given:** $\alpha_d{\cdot}\pi_d, R, R'$ *such that*

i) $\bot_S \notin I\,[\![\texttt{f:e}]\!]\ \alpha_d{\cdot}\pi_d\ R$

ii) *Safe-pat* $(\alpha{\cdot}\pi, \alpha_d{\cdot}\pi_d)$

iii) *Safe-comp-env* $(\rho)$

iv) *Equivalently-extended* $(\rho, R, R')$

**Need to show each of A,B and C below, which are exhaustive:**

$(I[\![\texttt{f:e}]\!]\ \alpha_d{\cdot}\pi_d\ R = I[\![\texttt{(fix:[f}\ \lambda\texttt{id.body])}\texttt{:e}]\!]\ \alpha_d{\cdot}\pi_d\ R')$ (A)

& *Safe-comp-env* $(\rho)$

if $(pa\ \alpha{\cdot}\pi) = unbound$ & $v\text{-}count \geq \iota$;

$(I[\![\texttt{f:e}]\!]\ \alpha_d{\cdot}\pi_d\ R = I[\![\texttt{(fix:[f}|\alpha{\cdot}\pi\ \lambda\texttt{id.body}_1\texttt{])}\texttt{:e}]\!]\ \alpha_d{\cdot}\pi_d\ R')$ (B)

& *Safe-comp-env* $(\rho_4)$

if $(pa\ \alpha{\cdot}\pi) = unbound$ & $v\text{-}count < \iota$;

$(I[\![\texttt{f:e}]\!]\ \alpha_d{\cdot}\pi_d\ R = I[\![\texttt{f}|\alpha{\cdot}\pi\texttt{:e}_1]\!]\ \alpha_d{\cdot}\pi_d\ R')$ (C)

& *Safe-comp-env* $(\rho_1)$, otherwise

**Proof:**

As shown in Case 3), there are three possible cases.

Case 11.1) $(\$ \notin \alpha_d{\cdot}\pi_d)$ & $(\$ \notin \alpha{\cdot}\pi)$; See Case 2).

Case 11.2) $(\$ \in \alpha_d{\cdot}\pi_d)$ & $(\$ \notin \alpha{\cdot}\pi)$; See Case 2).

Case 11.3) $(\$ \in \alpha_d{\cdot}\pi_d)$ & $(\$ \in \alpha{\cdot}\pi)$ There are three cases:

**Case 11.3.A**

$(\rho\ [\![\texttt{f}]\!])\downarrow 1\ =\ [\![\texttt{(fix:[f}\ \lambda\texttt{id.body])}]\!]$ (1)

By ($\mathcal{C}$ 11).

*Equivalently-extended* $(\rho, R, R')$ (2)

By iv).

$(I[\![\texttt{f:e}]\!]\ \alpha_d{\cdot}\pi_d\ R = I[\![\texttt{(fix:[f}\ \lambda\texttt{id.}\ \texttt{body])}\texttt{:e}]\!]\ \alpha_d{\cdot}\pi_d\ R)$

& *Safe-comp-env* $(\rho)$

By (1), (2), and iii).

**Case 11.3.B**

The argument is that for Case 9.3, with one exception. When $pat \neq \alpha{\cdot}\pi$, then $pa_1 = pa$, which is part of the safe environment $\rho$.

**Case 11.3.C**

$\perp_S \notin I[\![\mathtt{e}]\!]\ (\ [\![\mathtt{e}]\!] \blacksquare I[\![(\sqcup_{u=1}^{\infty}(U_u(\mathtt{fix:[f\ \lambda id.\ body]})))\mathtt{:e}]\!]\ \alpha_d{\cdot}\pi_d\ R)\ R$ (1)

By i) and a11).

$I[\![\mathtt{e}]\!]\ (\ [\![\mathtt{e}]\!] \blacksquare I[\![(\sqcup_{u=1}^{\infty}(U_u(\mathtt{fix:[f\ \lambda id.\ body]})))\mathtt{:e}]\!]\ \alpha_d{\cdot}\pi_d\ R)R\ =$
$I[\![\mathtt{e_1}]\!]\ (\ [\![\mathtt{e}]\!] \blacksquare I[\![(\sqcup_{u=1}^{\infty}(U_u(\mathtt{fix:[f|}\alpha{\cdot}\pi\ \mathtt{\lambda id.\ body_1]})))\mathtt{:e_1}]\!]\ \alpha_d{\cdot}\pi_d\ R')R'$
& *Safe-comp-env* $(\rho_1)$ (2)

Since $\alpha{\cdot}\pi$ has been seen before in the compilation of the outer *fix* expression in which $[\![\mathtt{f}]\!]$ was originally bound, $(pa\ \alpha{\cdot}\pi) \neq$ *unbound*. Note that the given invariant iii) shows that the pattern bound to $\alpha{\cdot}\pi$ in the environment $\rho$ is safely passed on to the compilation of $[\![\mathtt{e}]\!]$.

By (1), iii), iii) and IH.

$I[\![\mathtt{f:e}]\!]\ \alpha_d{\cdot}\pi_d\ R = I[\![\mathtt{f}|\alpha{\cdot}\pi\mathtt{:e_1}]\!]\ \alpha_d{\cdot}\pi_d\ R'$

By (2) and substitution. □

(C 12)

$C\ [\![\texttt{id}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota =$

$$\begin{cases} \textbf{Variable} & \text{if } b\text{-}type = \textbf{lambda}; \\ \textbf{Reached-Limit} & \text{if } (pa\ \alpha{\cdot}\pi) = unbound\ \&\ v\text{-}count \geq \iota; \\ \textbf{Compile-Binding} & \text{if } (pa\ \alpha{\cdot}\pi) = unbound\ \&\ v\text{-}count < \iota; \\ \textbf{Mark-With-Pattern} & \text{otherwise;} \end{cases}$$

**where** $\langle [\![\texttt{binding}]\!], pa, v\text{-}count, b\text{-}type\rangle = \rho\ [\![\texttt{id}]\!]$;

**Variable is**
$[\![\texttt{id}]\!]\ \alpha{\cdot}\pi\ \rho_1\ \iota$
**where**

$$\rho_1 = \lambda i. \begin{cases} \langle [\![\texttt{binding}]\!],\ (\alpha{\cdot}\pi \sqcup pa) \sqcap \pi_0, 0, b\text{-}type\rangle, \\ \quad \text{if } i = \texttt{id}; \\ \rho\, i, \text{otherwise;} \end{cases}$$

**Reached-Limit is** $[\![\texttt{binding}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota$;

**Compile-Binding is**
$[\![\texttt{fix:[id}|\alpha{\cdot}\pi\ \ \texttt{e}_1\texttt{]}]\!]\ \alpha{\cdot}\pi\ \rho_3\ \iota$
**where**
$[\![\texttt{fix:[id e]}]\!] = [\![\texttt{binding}]\!]$
$[\![\texttt{e}_1]\!]\ \alpha_1{\cdot}\pi_1\ \rho_1\ \iota_1 = C\ [\![\texttt{e}]\!]\ \alpha{\cdot}\pi\ \rho_2\ \iota$;

$$\rho_2 = \lambda i. \begin{cases} \langle [\![\texttt{fix:[id e]}]\!], pa_1, v\text{-}count + 1, \textbf{fix}\ \rangle, \\ \quad \text{if } i = [\![\texttt{f}]\!]; \\ \rho\, i, \text{otherwise;} \end{cases}$$

$$pa_1 = \lambda pat. \begin{cases} \bot & \text{if } pat = \alpha{\cdot}\pi; \\ unbound & \text{otherwise;} \end{cases}$$

$$\rho_3 = \lambda i. \begin{cases} \rho\, i, & \text{if } i = [\![\texttt{id}]\!]; \\ \rho_1\, i, & \text{otherwise.} \end{cases}$$

**Mark-With-Pattern is** $[\![\texttt{id}|\alpha{\cdot}\pi]\!]\ \alpha{\cdot}\pi\ \rho\ \iota$

**Case 12:** $[\![\texttt{exp}]\!] = [\![\texttt{id}]\!]$
**Given:** $\alpha_d{\cdot}\pi_d, R, R'$ *such that*
i) $\perp_S \notin I\ [\![\texttt{id}]\!]\ \alpha_d{\cdot}\pi_d\ R$
ii) *Safe-pat* $(\alpha{\cdot}\pi, \alpha_d{\cdot}\pi_d)$

iii) *Safe-comp-env* $(\rho)$

iv) *Equivalently-extended* $(\rho, R, R')$

**Need to show: All of A, B, C, D.** (A)

$(I[\![\texttt{id}]\!]\ \alpha_d{\cdot}\pi_d\ R = I[\![\texttt{id}]\!]\ \alpha_d{\cdot}\pi_d\ R')$ & *Safe-comp-env* $(\rho_1)$

if *b-type* = **lambda**

(B)

$(I[\![\texttt{id}]\!]\ \alpha_d{\cdot}\pi_d\ R = I[\![\texttt{fix:[f e]}]\!]\ \alpha_d{\cdot}\pi_d\ R')$

& *Safe-comp-env* $(\rho)$

if $(pa\ \alpha{\cdot}\pi) = unbound$ & *v-count* $\geq \iota$;

(C)

$I[\![\texttt{f:e}]\!]\ \alpha_d{\cdot}\pi_d\ R = I[\![\texttt{fix:[f e]}]\!]\ \alpha_d{\cdot}\pi_d\ R'$

& *Safe-comp-env* $(\rho_3)$

if $(pa\ \alpha{\cdot}\pi) = unbound$ & *v-count* $< \iota$;

(D)

$(I[\![\texttt{id}]\!]\ \alpha_d{\cdot}\pi_d\ R = I[\![\texttt{id}|\alpha{\cdot}\pi]\!]\ \alpha_d{\cdot}\pi_d\ R')$

& *Safe-comp-env* $(\rho)$, otherwise

**Proof:**

As shown in Case 3), there are three possible cases.

Case 12.1) $(\$ \notin \alpha_d{\cdot}\pi_d)$ & $(\$ \notin \alpha{\cdot}\pi)$; See Case 2).

Case 12.2) $(\$ \in \alpha_d{\cdot}\pi_d)$ & $(\$ \notin \alpha{\cdot}\pi)$; See Case 2).

Case 12.3) $(\$ \in \alpha_d{\cdot}\pi_d)$ & $(\$ \in \alpha{\cdot}\pi)$

There are four cases:

**Case 12.3.A** (1)

$(\rho_1\ [\![\texttt{id}]\!]) \neq unbound$ By (*C* 12).

(2)

(*Binding-type* $(\rho_1\ [\![\texttt{id}]\!])) =$ **lambda**

$\forall\ R, \alpha_d{\cdot}\pi_d, \rho'$ *such that* *Equivalently-extended* $(\rho', R)$, *Extension-of* $(\rho, \rho')$,

$\forall\ [\![\mathtt{E}]\!]$ *such that* $[\![\mathtt{E}]\!] = [\![(\lambda \mathtt{id.body}) : \mathtt{e}]\!]$

$pa = (\ [\![\mathtt{id}]\!]_i$ *where* $1 \leq i < n) \bullet C\ [\![\mathtt{E}]\!]\ \alpha{\cdot}\pi\ \rho'\ \iota$

&

$(\alpha{\cdot}\pi \sqcup pa) = (\ [\![\mathtt{id}]\!]_{i+1}$ *where* $1 \leq i < n) \bullet C\ [\![\mathtt{E}]\!]\ \alpha{\cdot}\pi\ \rho'\ \iota$

&

$(\alpha{\cdot}\pi \sqcup pa) \sqcap \pi_0 \sqsubseteq (\ [\![\mathtt{id}]\!]_n$ *where* $1 \leq i \leq n) \bullet C\ [\![\mathtt{E}]\!]\ \alpha{\cdot}\pi\ \rho'\ \iota$ (3)

(4)

*Safe-lambda-exp* $(\rho_1, [\![\mathtt{id}]\!])$ By (1), (2) and (3).

$\forall\ [\![\mathtt{ide}]\!]$

$(\rho_1\ [\![\mathtt{ide}]\!]) \neq$ *unbound* $\Longrightarrow$

$((\mathit{Binding\text{-}type}\ (\rho_1\ [\![\mathtt{ide}]\!])) = \mathbf{lambda}) \Longrightarrow$ *Safe-lambda-exp* $(\rho_1, [\![\mathtt{ide}]\!])$ (5)

By iii) and (4).

(6)

*Safe-comp-env* $(\rho_1)$ By iii) and (5).

**Case 12.3.B**

(1)

$(\rho\ [\![\mathtt{id}]\!])\downarrow 1 = [\![\mathtt{fix:[id\ e]}]\!]$ By ($\mathcal{C}$ 12).

(2)

*Equivalently-extended* $(\rho, R, R')$ By iv).

$(I[\![\mathtt{id}]\!]\ \alpha_d{\cdot}\pi_d\ R = I[\![\mathtt{fix:[id\ e]}]\!]\ \alpha_d{\cdot}\pi_d\ R')$

& *Safe-comp-env* $(\rho)$

By (1), (2), substitution and iii).

**Case 12.3.C**

The argument is that for Case 10.3, with one exception. When $pat \neq \alpha{\cdot}\pi$, then $pa_1 = pa$, which is part of the safe environment $\rho$.

**Case 12.3.D**

At this point, $[\![\mathtt{id}|\alpha\cdot\pi]\!]$ has been introduced correctly in the surrounding *fix* expression; this labeling ensures that it will refer to the outer binding properly. □ ■

## Chapter 4: Further analysis of conditional expressions

This chapter presents some improvements upon the equation for **if** given in Chapter 2, which is very simple but not powerful enough to handle some common and useful programming styles. Functions written in iterative style are discussed first, and then functions which process potentially finite lists. Finally, the two are combined.

### Section 4.1: Iterative functions

A function is in *iterative* style if it can be written as

```
⟦F = λz. if:<p:z f:z F:g:z>⟧.                    (1)
```

where ⟦p⟧, ⟦f⟧ and ⟦g⟧ are primitive functions or the composition of primitive functions. Like iterative loops, functions written in iterative style can be implemented using tail recursion rather than a stack.

For example, the function ⟦Rev⟧, defined as

```
⟦Rev =
 λz.                                             (2)
   if:<nil?:head:z
       head:tail:z
       Rev:<tail:head:z <head:head:z . head:tail:z>>>⟧.
```

constructs a reversed sublist, ⟦head:tail:z⟧, that eventually contains the complete reverse of list ⟦head:z⟧. The factorial function ⟦Fact2⟧, written as

```
⟦Fact2 =
 λz.                                                          (3)
   if:<zero?:head:z
        head:tail:z
        Fact2:<dcr:head:z mpy:<head:z head:tail:z>>>⟧.
```

multiplies ⟦head:z⟧ by ⟦head:tail:z⟧, an accumulated value which eventually becomes the factorial of ⟦head:z⟧.

Consider the compilation of an application of ⟦Fact2⟧ to the argument ⟦<2 1>⟧. Compilation of an iterative function's application must synthesize a pattern for ⟦z⟧ which is a solution to the equation

$$\begin{aligned} &[\![z]\!] \bullet C\,[\![F{:}args]\!]\ \alpha{\cdot}\pi\ \rho\ \iota = \\ &\quad [\![z]\!] \bullet C\,[\![p{:}z]\!]\ \alpha{\cdot}\pi'\ \rho\ \iota \ \sqcup \\ &\quad ([\![z]\!] \bullet C\,[\![f{:}z]\!]\ \alpha{\cdot}\pi\ \rho'\ \iota \ \sqcap \ [\![z]\!] \bullet C\,[\![F{:}g{:}z]\!]\ \alpha{\cdot}\pi\ \rho'\ \iota). \end{aligned} \tag{4}$$

The function • has been defined operationally, and so its codomain is a subset of rational patterns. The object of a search for a fixpoint of (4) is the most powerful (most strict) member of a set of candidate fixpoints. This set must contain the lazy pattern, as that may be the only possible solution.

At first glance, the incomplete sublattice from Chapter 2 appears to be a good set of candidate patterns. Unfortunately, a simple unbounded search will not terminate because an infinite number of cyclic patterns must be explored, even if the acyclic ones are ruled out. One possible approach is to restrict the search to exploration of a limited number of patterns. Given the unpredictable distribution of the possible solutions in the lattice and the immense size of the lattice, one cannot argue that this naive approach is likely to terminate with any success.

Fortunately, some additional information can be brought to bear on the problem. $(\llbracket \mathtt{z} \rrbracket \bullet C\ \llbracket \mathtt{p:z} \rrbracket\ \alpha{\cdot}\pi'\ \rho\ \iota) \sqcup (\llbracket \mathtt{z} \rrbracket \bullet C\ \llbracket \mathtt{f:z} \rrbracket\ \alpha{\cdot}\pi\ \rho'\ \iota)$ can only be a cyclic (rational) or finite pattern, since $\llbracket \mathtt{f} \rrbracket$ is a primitive function and the outer call to $\llbracket \mathtt{F} \rrbracket$ inherits a rational pattern. In addition, $\llbracket \mathtt{z} \rrbracket \bullet C\ \llbracket \mathtt{F:args} \rrbracket\ \alpha{\cdot}\pi\ \rho\ \iota \sqsubseteq (\llbracket \mathtt{z} \rrbracket \bullet C\ \llbracket \mathtt{p:z} \rrbracket\ \alpha{\cdot}\pi'\ \rho\ \iota) \sqcup (\llbracket \mathtt{z} \rrbracket \bullet C\ \llbracket \mathtt{f:z} \rrbracket\ \alpha{\cdot}\pi\ \rho'\ \iota)$. Instead of constraining the search strategy directly, it seems reasonable to develop a finite lattice which contains $(\llbracket \mathtt{z} \rrbracket \bullet C\ \llbracket \mathtt{p:z} \rrbracket\ \alpha{\cdot}\pi'\ \rho\ \iota) \sqcup (\llbracket \mathtt{z} \rrbracket \bullet C\ \llbracket \mathtt{f:z} \rrbracket\ \alpha{\cdot}\pi\ \rho'\ \iota)$ as its top element; this pattern is the best that can be hoped for, and so is called the *target* pattern. This lattice can then be searched, exhaustively if necessary; an algorithm that performs this search is discussed later in this chapter.

Such a finite lattice is developed by constructing a homomorphism $H$ from the complete infinite lattice $P$ to a finite lattice $P_{finite,n,\pi}$. In order to present this homomorphism, the following terms are introduced:

— An *irrational* pattern is an infinite pattern that cannot be finitely and explicitly represented as a cyclic graph.

— A *fully repeating* pattern is a rational pattern that can be represented by a cyclic graph containing cyclic references which refer only to the entire pattern. An example of such a pattern is

$$fix\,\lambda\pi.\langle \$\bot\ ,\ \pi\rangle.$$

— A *partially repeating* pattern is a rational pattern which can be represented by a cyclic graph containing cyclic references which may refer to sub-patterns within the entire pattern. An example of a partially repeating pattern is

$$\langle \bot\ ,\ \$fix\,\lambda\pi.\langle \$\pi\ ,\ \pi\rangle\rangle.$$

— A *finite* pattern can be represented without any cycles. An example of a finite pattern is

$$\langle \bot\ ,\ \$\bot\rangle.$$

— An *n-bounded* pattern is a fully repeating rational or finite pattern whose maximum cycle length is less than or equal to $n$. For example,

$$\langle \$\bot \ , \ \langle \$\bot \ , \ \bot \rangle \rangle$$

is 2-*bounded* but not 1-*bounded.* Similarly,

$$fix\lambda\pi.\langle \bot \ , \ \langle \pi \ , \ \langle \bot \ , \ \pi \rangle \rangle \rangle$$

is 4-*bounded* but not 2-*bounded.*

**Definition:** A finite pattern p represents the *truncation* of a pattern q, written as $(Trunc\ q\ n)$, if p and q are identical except that at some node in the representations of p and q within the bound $n$, p contains $\bot_p$ where q contains a pattern above $\bot_p$.

Put very simply, the desired homomorphism $H$ selects n-bounded patterns. The effect that this bound $n$ has upon the selection of patterns from $P_{finite,n,\pi}$ can be more easily understood if the domain of patterns is partitioned as follows:

1) Irrational patterns
2) Rational patterns
   a) Fully repeating patterns
      i) $n$-bounded fully repeating patterns
      ii) fully repeating patterns with bound greater than $n$
   b) Partially repeating patterns
   c) Finite patterns
      i) $n$-bounded finite patterns
      ii) finite patterns with bound greater than $n$.

$H$ maps all patterns in 2.a.i. and 2.c.i. to themselves. All others are mapped to finite *truncations* of themselves. A more formal definition follows:

**Definition:** $H : P \times INT \longrightarrow P =$

$\lambda\ p\ n.$

$irrational?\ p \longrightarrow Trunc\ p\ n;$

$fully - repeating?\ p \longrightarrow$

$n - bounded?\ p \longrightarrow p;\ Trunc\ p\ n;$

$partially - repeating?\ p \longrightarrow Trunc\ p\ n;$

$n - bounded?\ p \longrightarrow p;\ Trunc\ p\ n$

**Definition:** $P_{finite,n,\pi}$ is the image of $P$ under $H$, some $n : INT$ and some $\pi : P$.

Using truncation, it is possible to form a finite approximation to all infinite and finite structures within $P$, as shown in the following theorem.

**Theorem 4.1:** $P_{finite,n,\pi}$ is a finite lattice.

**Proof:** $H$ maps infinite patterns either into themselves or a pattern whose depth has the finite bound, $n$. $H$ maps finite patterns either into themselves, if their depth is smaller than or equal to n, or into a truncated pattern whose depth has a finite bound. ■

**Theorem 4.2:** $(Trunc\ \pi\ n) \sqsubseteq_P \pi$ in $P$.

**Proof:** Since $\pi$'s truncation is formed by substituting $\perp_P$ for some pattern equal to or higher than $\perp_P$ in $\pi$, $(Trunc\ \pi\ n)$ is under or equal to $\pi$ in $P$. ■

Theorem 4.2 makes it easy to see that different finite lattices formed by $H$ from $\pi$ with different bounds $n$ have different expressive powers, which can be given an ordering based upon the size of the bound used to construct each one. For example, consider a very simple lattice $\pi$ with bound 2 (written as $P_{finite,2,\pi_0}$). It is possible to see the beginning of an infinite sequence of patterns approximating $fix\,\lambda\pi.\langle\$\pi\ ,\ \pi\rangle$ from below but never actually reaching $fix\,\lambda\pi.\langle\$\pi\ ,\ \pi\rangle$. The interesting infinite patterns of $P_{finite,2,\pi_0}$ are included in those of $P_{finite,3,\pi_0}$ and in addition, there are new cyclic patterns, including $fix\,\lambda\pi.\langle\$\langle\$\perp\ ,\ \perp\rangle\ ,\ \pi\rangle$, an approximation to $fix\,\lambda\pi.\langle\$\pi\ ,\ \pi\rangle$.

It is now possible to discuss techniques for sifting through candidate patterns in any $P_{finite,n,\pi}$ in order to find a solution to the equation

$$\begin{aligned}&[\![\mathtt{z}]\!] \bullet C\,[\![\mathtt{F{:}args}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota = \\ &\quad [\![\mathtt{z}]\!] \bullet C\,[\![\mathtt{p{:}z}]\!]\ \alpha{\cdot}\pi'\ \rho\ \iota\ \sqcup \\ &\quad (\,[\![\mathtt{z}]\!] \bullet C\,[\![\mathtt{f{:}z}]\!]\ \alpha{\cdot}\pi\ \rho'\ \iota \sqcap [\![\mathtt{z}]\!] \bullet C\,[\![\mathtt{F{:}g{:}z}]\!]\ \alpha{\cdot}\pi\ \rho'\ \iota).\end{aligned} \tag{4}$$

Truncation permits the construction of a set of "weaker" patterns from the target pattern, called the *weaker set* for this pattern. This set becomes the set of lattice points immediately beneath this initial pattern. If no element of this set satisfies the equations, then each may in its turn be used to create a subsequent set of even weaker patterns, and the process may be continued until the lattice has been searched completely, at which point $\bot\mathtt{p}$ is the only possible solution. The algorithm used to construct lattices of finite patterns from the top element downwards can be expressed as follows:

If a pattern $\pi$ has $m$ leaf and interior nodes in its representation, then there are a total of $m$ possible elements of the weaker set for $\pi$, one at each node. $m$ copies of $\pi$ are created, each of which has been altered at a unique node, following these three rules:

— the weakening of $\$\langle\pi_1\ ,\ \pi_2\rangle$ is $\langle\pi_1\ ,\ \pi_2\rangle$;

— the weakening of $\langle\bot\ ,\ \bot\rangle$ is $\bot$;

— the weakening of $\$\bot$ is $\bot$.

If, during the creation process, the sub-pattern at this unique node cannot be altered by one of these rules (if it is $\bot$, for example), then it is not included among the weaker set for $\pi$.

A simple example follows:

Suppose the target pattern is $\$\langle\$\bot\ ,\ \$\bot\rangle$,

— initial set is $\{ \langle \$\bot , \$\bot \rangle_1 \$\langle \bot , \$\bot \rangle_2, \$\langle \$\bot , \bot \rangle_3 \}$;
— set for $\langle \$\bot , \$\bot \rangle_1$ is $\{ \langle \bot , \$\bot \rangle_{11}, \langle \$\bot , \bot \rangle_{12} \}$;
— set for $\$\langle \bot , \$\bot \rangle_2$ is $\{ \langle \bot , \$\bot \rangle_{21}, \$\langle \bot , \bot \rangle_{22} \}$;
— set for $\$\langle \$\bot , \bot \rangle_3$ is $\{ \langle \$\bot , \bot \rangle_{31}, \$\langle \bot , \bot \rangle_{32} \}$;

— set for $\langle \bot , \$\bot \rangle_{11}$ is $\{ \langle \bot , \bot \rangle_{111} \}$;
— set for $\langle \$\bot , \bot \rangle_{12}$ is $\{ \langle \bot , \bot \rangle_{121} \}$;
— set for $\langle \bot , \$\bot \rangle_{21}$ is $\{ \langle \bot , \bot \rangle_{211} \}$;
— set for $\$\langle \bot , \bot \rangle_{22}$ is $\{ \$\bot_{221} \}$;
— set for $\langle \$\bot , \bot \rangle_{31}$ is $\{ \langle \bot , \bot \rangle_{311} \}$;
— set for $\$\langle \bot , \bot \rangle_{32}$ is $\{ \$\bot_{321} \}$;

— set for $\langle \bot , \bot \rangle_{111}$ is $\{ \bot \}$;
— set for $\langle \bot , \bot \rangle_{121}$ is $\{ \bot \}$;
— set for $\langle \bot , \bot \rangle_{211}$ is $\{ \bot \}$;
— set for $\$\bot_{221}$ is $\{ \bot \}$;
— set for $\langle \bot , \bot \rangle_{311}$ is $\{ \bot \}$;
— set for $\$\bot_{321}$ is $\{ \bot \}$.

Weakening a cyclic pattern, $\pi$, is a slightly more complex process. The pattern is first unrolled until truncation prevents further expansion. Cycles interrupted by the truncation are replaced by $\bot$ at the point the expansion crosses the bound $n$. This ensures that all possible cyclic patterns within $n$ nodes from the root will appear in the weaker set for $\pi$. (For example, the weaker set for the pattern $fix\ \lambda\ \pi.\langle \$\bot , \pi \rangle$ when $n$ is 4 must include patterns such as $fix\ \lambda\ \pi.\langle \$\bot , \langle \bot , \langle \bot , \pi \rangle \rangle \rangle$.) Next, $m$ copies are made, one for each node in the cyclic tree representing $\pi$. Each copy is altered at a unique node according to the rules presented, unless the sub-pattern present at the node is either $\bot$ or a cyclic reference. A new rule to generate weakenings for cyclic references is added:

— the weakening of *cyclic reference* is $\bot$.

All possible cyclic patterns immediately below $\pi$, as well as the truncations of all finite approximations of $\pi$, have been accumulated at the end of this process.

Weakening is a simple and efficient way of identifying all possible paths from the top of the lattice down to the lowest element in the lattice. Weakenings produced from a pattern form a set of incomparable patterns, however a set of weakened patterns can include patterns that appear in other sets. These duplicates represent lattice nodes with out-degree of two or higher, and can easily be removed for the sake of efficiency.

### 4.1.1: Iterative style equation

The following equation is the equation for **if** from Chapter 2 modified so that it accommodates iterative style.

The compilation of the second branch of the **if** expression requires some explanation. Instead of a simple compilation of the expression $[\![\mathtt{F:g:z}]\!]$, this simple compilation is embedded in a loop which tests each element of the weaker set for the target pattern, recursively creating and testing weaker sets for each of these elements until one succeeds. Success occurs when the weakening of the target pattern inherited by the compilation of $[\![\mathtt{g:z}]\!]$ is an equivalent pattern to $[\![\mathtt{z}]\!] \bullet C\ [\![\mathtt{F:g:z}]\!]\ \alpha{\cdot}\pi\ \rho'\ \iota$, thus providing a solution to Equation (4). $\perp$ will succeed when all other patterns fail.

*New-w* produces a set of weakened patterns from its argument pattern, and is not explicitly described here beyond the algorithm previously discussed.

$$
\begin{array}{l}
C\ [\![\texttt{if:<p:z f:z F:g:z>}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota \\
\textbf{where}\ (\textit{Binding-type}\ (\rho\ [\![\texttt{F}]\!])) = \textbf{fix}, \\
\quad (\textit{Binding-type}\ (\rho\ [\![\texttt{z}]\!])) = \textbf{lambda}, \\
\quad (\$ \in \alpha{\cdot}\pi) \\
= \\
[\![\texttt{if:<\$e1}_1\ \texttt{e2}_2\ \texttt{F}|\alpha{\cdot}\pi\texttt{:e3}_3\texttt{>}]\!]\ \alpha{\cdot}\pi\ \rho_3\ \iota \\
\textbf{where} \\
\quad [\![\texttt{e1}_1]\!]\ \pi_1\ \rho_1\ \iota = C\ [\![\texttt{p:z}]\!]\ \$\bot\ \rho\ \iota \\
\quad [\![\texttt{e2}_2]\!]\ \pi_2\ \rho_2\ \iota = C\ [\![\texttt{f:z}]\!]\ \alpha{\cdot}\pi\ \rho_1\ \iota \\
\quad [\![\texttt{e3}_3]\!]\ \pi_3\ \rho_3\ \iota = \\
\quad (\lambda\ cw. \\
\quad\quad (\textit{fix}\ \lambda\ \Psi.\lambda\ cw\ copy. \\
\quad\quad\quad \left\{ \begin{array}{l}
[\![\texttt{g:z}]\!]\ \alpha{\cdot}\pi\ \rho_1\ \iota, \\
\qquad \text{if } cw = () \ \&\ (\textit{New-w}\ copy) = (); \\
\Psi\ (\textit{New-w}\ copy)\ (\textit{New-w}\ copy), \\
\qquad \text{if } cw = () \ \&\ (\textit{New-w}\ copy) \neq (); \\
[\![\texttt{e}_t]\!]\ \pi_t\ \rho_t\ \iota \\
\textbf{where}\ [\![\texttt{e}_t]\!]\ \pi_t\ \rho_t\ \iota = \\
\quad C\ [\![\texttt{g:z}]\!]\ (cw{\downarrow}1)\ \rho\ \iota, \\
\qquad \text{if } (\textit{Pat-fun}\ (\rho_t\ [\![\texttt{z}]\!])) = (cw{\downarrow}1); \\
\Psi\ (cw{\downarrow}2)\ copy, \\
\qquad \text{otherwise}
\end{array} \right. \\
\quad\quad )\ cw\ cw)\ (\pi_{new}, (\textit{New-w}\ \pi_{new})) \\
\quad \textbf{where} \\
\quad \pi_{new}\ =\ (\textit{Pat-fun}\ (\rho_2\ [\![\texttt{z}]\!]))
\end{array}
$$

(The projection functions *Binding-type* and *Pat-fun* have been defined in Chapter 2. These functions select elements of the values bound to variables in the compile-time environment.)

### 4.1.2: Example

The following application of $[\![\texttt{Fact2}]\!]$ is compiled with the strictness pattern $\$\bot$:

```
⟦(fix:[F
 λz.
   if:<zero?:head:z
       head:tail:z
       F:<dcr:head:z .
            <mpy:<head:z head:tail:z> .  <>>>>])
 :<a .  <b .  <>>>⟧
```

$\llbracket \texttt{z} \rrbracket \bullet C \llbracket \texttt{zero?:head:z} \rrbracket \ \$\bot \ \rho_{init} \ 3 = \$\langle \$\bot , \bot \rangle$,

$(\llbracket \texttt{z} \rrbracket \bullet C \llbracket \texttt{zero?:head:z} \rrbracket \ \$\bot \ \rho_{init} \ 3) \sqcup (\llbracket \texttt{z} \rrbracket \bullet C \llbracket \texttt{head:tail:z} \rrbracket \ \$\bot \ \rho_1 \ 3)$

$= \$\langle \$\bot , \bot \rangle \sqcup \$\langle \bot , \$\langle \$\bot , \bot \rangle \rangle \sqcap \pi_0$

$= \$\langle \$\bot , \langle \$\bot , \bot \rangle \rangle$.

At this point, the initial weakening of the target pattern is tested:

$\llbracket \texttt{<dcr:head:z mpy:<head:z head:tail:z>>} \rrbracket$

$\bullet C \llbracket \texttt{F:<dcr:head:z mpy:<head:z head:tail:z>>} \rrbracket \ \$\bot \ \rho_1 \ 3 =$

$\$\langle \$\bot , \langle \$\bot , \bot \rangle \rangle$.

$\llbracket \texttt{z} \rrbracket \bullet C \llbracket \texttt{F:<dcr:head:z mpy:<head:z head:tail:z>>} \rrbracket \ \$\bot \ \rho_1 \ 3 =$

$\$\langle \$\bot , \bot \rangle \sqcup \$\langle \$\bot , \bot \rangle \sqcup \$\langle \bot , \$\langle \$\bot , \bot \rangle \rangle \sqcap \pi_0$

$= \$\langle \$\bot , \langle \$\bot , \bot \rangle \rangle$

As

$\llbracket \texttt{z} \rrbracket \bullet C \llbracket \texttt{F:<dcr:head:z mpy:<head:z head:tail:z>>} \rrbracket \ \$\bot \ \rho_1 \ 3 =$

$\llbracket \texttt{<dcr:head:z mpy:<head:z head:tail:z>>} \rrbracket$

$\bullet C \llbracket \texttt{F:<dcr:head:z mpy:<head:z head:tail:z>>} \rrbracket \ \$\bot \ \rho_1 \ 3$,

the final result of compilation is

```
[[(fix:[F|$⊥
 λz.
   if:<$zero?:head:z
         head:tail:z
         F|$⊥:<$dcr:head:z .
           <mpy:<$head:z $head:tail:z> . $<>>>>])
:<$a . <$b . $<>>>]]
```

## Section 4.2: List mapping functions

Functions in which [[p:z]] is the application of [[nil]] to some composition of the primitive functions [[head]] and [[tail]] are a very common and useful way of constructing new lists in lazy languages. A well known example is the Lisp mapping function. Similar *list mapping* functions are considered here written in the following form:

```
[[F = λz. if:<p:z nil M:F:g:z>]].
```

This can be perceived as similar to the syntax given for iterative functions in the previous section. [[f]] is the constant function [[nil]], [[g]] is a primitive function or the composition of primitive functions, and [[M]] may be primitive or user-defined.

As an example, consider the vector increment function [[Vinc]],

```
⟦Vinc =
 λz.
    if:<nil?:head:z
          nil
          <inc:head:head:z . Vinc:<tail:head:z>>>⟧.
```

⟦Vinc⟧ is rewritten so that the substructure of ⟦z⟧ is named explicitly:

```
⟦Vinc =
 λ[lst].
  if:<nil?:lst
        nil
        <inc:head:lst . Vinc:<tail:lst>>>⟧.
```

So far, this analysis has not permitted us to mark the recursion on ⟦tail:lst⟧. The **if** equation presented in Chapter 2 causes any variable that appears on only one branch of the **if** to inherit a pattern of $\perp_p$, as the *meet* is taken of variables appearing in both branches, and the initial pattern inherited by a variable is assumed to be $\perp_p$. A change in this equation is required, if the construction of ⟦lst⟧ is to be made strict in its inner structure. This change can be developed from the current equation in the following way, using an example to clarify the problem.

Suppose a call to ⟦Vinc⟧ inherits the pattern $fix\,\lambda\pi.\langle \$\perp , \pi\rangle$. The ⟦nil⟧ test propagates information only about the outer structure of its argument, and does not justify any marking of the inner structure. However, the pattern inherited from both branches becomes $\perp_p$, so no information about ⟦lst⟧'s inner structure becomes available to the analysis of the expression that constructs ⟦lst⟧.

Fortunately, the presence of a ⟦nil?⟧ test provides us with some useful information about ⟦lst⟧. The following theorem permits the analysis of such a loop to use only the pattern propagated by the recursive branch, so that the pattern inherited by ⟦lst⟧ becomes $\$fix\,\lambda\pi.\langle\$\bot\ ,\ \pi\rangle$.

**Theorem 4.3** In a **list mapping** function application containing a ⟦nil?⟧ test on list ⟦lst⟧, ⟦lst⟧ 's inherited pattern is the *Join* of the patterns propagated to ⟦lst⟧ by the analysis of the predicate and the looping branch.

**Proof:** The non-trivial case occurs when the loop inherits a list pattern. When initially entering a loop at runtime with a ⟦nil?⟧ test on ⟦lst⟧, there are two possibilities: either ⟦lst⟧ is nil, or it is non-nil and has some internal structure.

If the list ⟦lst⟧ is empty on first entering the loop, then the expression bound to ⟦lst⟧ must have produced this value. But, if it did so, then it did not execute any code building up the interior of the list. Thus, no unnecessary runtime divergence is introduced by marking this code, because it wasn't executed.

If ⟦lst⟧ has some internal structure at run time, then it is appropriate to propagate a pattern that specifies strictness within this structure. ■

### 4.2.1: List mapping function equation

The following equation is the equation for **if** from Chapter 2 developed for higher order functions, and modified so that it accounts for ⟦nil?⟧ tests on an argument. The essential alteration is derived from Theorem 4.3.

$C$ ⟦if:<nil?:p:z nil M:F:g:z>⟧ $\alpha{\cdot}\pi\ \rho\ \iota$
**where** (*Binding-type* ($\rho$ ⟦F⟧)) = **fix**,
(*Binding-type* ($\rho$ ⟦z⟧)) = **lambda**,
($\$ \in \alpha{\cdot}\pi$)
=
⟦if:<$e1$_1 e2$_2$ e3$_3$ >⟧ $\alpha{\cdot}\pi\ \rho_4\ \iota$
**where**
⟦e1$_1$⟧ $\pi_1\ \rho_1\ \iota_1 = C$ ⟦nil?:p:z⟧ $\$\bot\ \rho\ \iota$
⟦e2$_2$⟧ $\pi_2\ \rho_2\ \iota_2 = C$ ⟦nil⟧ $\alpha{\cdot}\pi\ \rho_1\ \iota$
⟦e3$_3$⟧ $\pi_3\ \rho_3\ \iota_3 = C$ ⟦M:F:g:z⟧ $\alpha{\cdot}\pi\ \rho_1\ \iota$;

$$\rho_4 = \lambda i. \begin{cases} \langle [\![\text{binding}_2]\!], (pa_1 \sqcup pa_3), 0, b\text{-}type_2\rangle \\ \quad \text{if } b\text{-}type_2 = \textbf{lambda} \ \&\ i = [\![\text{z}]\!]; \\ \langle [\![\text{binding}_2]\!], (pa_2 \sqcap pa_3), 0, b\text{-}type_2\rangle \\ \quad \text{if } b\text{-}type_2 = \textbf{lambda}; \\ \langle [\![\text{binding}_2]\!], pa_4, v\text{-}count_2 + v\text{-}count_3, b\text{-}type_2\rangle \\ \quad \text{if } b\text{-}type_2 = \textbf{fix}\ ; \end{cases}$$

**where**
$\langle [\![\text{binding}_1]\!], pa_1, v\text{-}count_1, b\text{-}type_1\rangle = \rho_1\ i$
$\langle [\![\text{binding}_2]\!], pa_2, v\text{-}count_2, b\text{-}type_2\rangle = \rho_2\ i$
$\langle [\![\text{binding}_3]\!], pa_3, v\text{-}count_3, b\text{-}type_3\rangle = \rho_3\ i$

$$pa_4 = \lambda pat. \begin{cases} (pa_3\ pat) & \text{if } (pa_2\ pat) = unbound; \\ (pa_2\ pat) & \text{otherwise.} \end{cases}$$

### 4.2.2: Example

Consider the compilation of the following application of ⟦Vinc⟧, with the strictness pattern *fix* $\lambda\ \pi.\langle \$\bot\ ,\ \pi\rangle$ :

```
⟦(fix:[Vinc
 λz.
   if:<nil?:head:z
         nil
         <inc:head:head:z . Vinc:<tail:head:z>>>>]):
<<a . <b . <c . <>>>>>⟧.
```

$[\![\mathtt{z}]\!] \bullet C\,[\![\mathtt{nil?:head:z}]\!]\ \mathit{fix}\,\lambda\pi.\langle \$\bot\ ,\ \pi\rangle\ \rho_{init}\ 4$

$= \$\langle \$\bot\ ,\ \bot\rangle.$

For the rest of this example,

$vincsyn\ =\ ((Pat\text{-}fun\ (\rho\ [\![\mathtt{Vinc}]\!]))\ \mathit{fix}\,\lambda\pi.\langle \$\bot\ ,\ \pi\rangle).$

$([\![\mathtt{z}]\!] \bullet C\,[\![\mathtt{nil?:head:z}]\!]\ \mathit{fix}\,\lambda\pi.\langle \$\bot\ ,\ \pi\rangle\ \rho_{init}\ 4) \sqcup$
$([\![\mathtt{z}]\!] \bullet C\,[\![\mathtt{<inc:head:head:z\ .\ Vinc:<tail:head:z>>}]\!]\ \mathit{fix}\,\lambda\pi.\langle \$\bot\ ,\ \pi\rangle\ \rho'\ 4) =$
$\$\langle \$\bot\ ,\ \bot\rangle \sqcup \$\langle \$\langle \$\bot\ ,\ \bot\rangle\ ,\ \bot\rangle \sqcup \langle\langle \bot\ ,\ vincsyn{\downarrow}1\rangle\ .\ \bot\rangle$
$= \$\langle \$\langle \$\bot\ .\ vincsyn{\downarrow}1\rangle\ .\ \bot\rangle.$ (5)

$vincsyn{\downarrow}1\ =\ \$\langle \$\bot\ ,\ vincsyn{\downarrow}1\rangle.$

(5) can be written as $\mathit{fix}\ \lambda\pi\ .\$\langle \$\bot\ ,\ \pi\rangle.$

Since $(\mathit{fix}\ \lambda\pi\ .\$\langle \$\bot\ ,\ \pi\rangle \sqcap \pi_0) = \$\mathit{fix}\,\lambda\pi.\langle \$\bot\ ,\ \pi\rangle,$

$[\![\mathtt{z}]\!] \bullet C\,[\![\mathtt{Vinc:<<a\ .\ <b\ .\ <c\ .\ <>>>>>}]\!]\ \mathit{fix}\,\lambda\pi.\langle \$\bot\ ,\ \pi\rangle\ \rho_{init}\ 4 =$
$\$\langle \$\mathit{fix}\,\lambda\pi.\langle \$\bot\ ,\ \pi\rangle\ ,\ \bot\rangle.$

The result of the compilation is

```
[(fix:[Vinc|fix λπ.⟨$⊥ , π⟩
 λz.
   if:<$nil?:head:z
         nil
         <$inc:head:head:z .
         Vinc|fix λπ.⟨$⊥ , π⟩:<$tail:head:z>>>]):
 <$<$a . <$b . <$c . $<>>>>>>].
```

## Section 4.3: Combining iterative and mapping functions

The combination of the two equations presented for **if** in this chapter permit us to analyze functions such as ⟦Rev⟧, which both *tests* its list argument and *accumulates* the reversed result.

### 4.3.1: Iterative mapping function equation

The following equation is the equation presented for iterative loops, except that the target pattern is produced slightly differently. The *Join* of the patterns inherited by ⟦z⟧ during compilation of the predicate and second branch are 'or'ed into the target pattern. *Mask* selects the sub-pattern of ⟦F⟧'s synthesized pattern to be protected, then *Surround* builds up a surrounding pattern that contains only $\perp$, and so will not affect the *Join*. The mechanism used to create this pattern is complicated by the fact that the user is permitted to specify the sub-list rather than be restricted to a limiting syntax; thus the trajectory (location within the argument tree structure) of the sub-list must be determined at compile time.

$C\ [\![\texttt{if:<nil?:p:z f:z F:g:z>}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota$
**where** $(\textit{Binding-type}\ (\rho\ [\![\texttt{F}]\!])) = \textbf{fix}$,
$(\textit{Binding-type}\ (\rho\ [\![\texttt{z}]\!])) = \textbf{lambda}$,
$(\$ \in \alpha{\cdot}\pi)$
$=$
$[\![\texttt{if:<\$e1}_1\ \texttt{e2}_2\ \texttt{F}|\alpha{\cdot}\pi : \texttt{e3}_3\texttt{>}]\!]\ \alpha{\cdot}\pi\ \rho_3\ \iota$
**where**
$[\![\texttt{e1}_1]\!]\ \pi_1\ \rho_1\ \iota = C\ [\![\texttt{nil?:p:z}]\!]\ \$\bot\ \rho\ \iota$
$[\![\texttt{e2}_2]\!]\ \pi_2\ \rho_2\ \iota = C\ [\![\texttt{f:z}]\!]\ \alpha{\cdot}\pi\ \rho_1\ \iota$
$[\![\texttt{e3}_3]\!]\ \pi_3\ \rho_3\ \iota =$
$(\lambda\ cw.$
$(\textit{fix}\ \lambda\ \Psi.\lambda\ cw\ copy.$
$\begin{cases} [\![\texttt{g:z}]\!]\ \alpha{\cdot}\pi\ \rho_1\ \iota, \\ \qquad \text{if } cw = () \ \&\ (\textit{New-w}\ copy) = (); \\ \Psi\ (\textit{New-w}\ copy)\ (\textit{New-w}\ copy), \\ \qquad \text{if } cw = () \ \&\ (\textit{New-w}\ copy) \neq (); \\ [\![\texttt{e}_t]\!]\ \pi_t\ \rho_t\ \iota \\ \textbf{where}\ [\![\texttt{e}_t]\!]\ \pi_t\ \rho_t\ \iota = \\ \quad C\ [\![\texttt{g:z}]\!]\ (cw{\downarrow}1)\ \rho\ \iota, \\ \qquad \text{if } (\textit{Pat-fun}\ (\rho_t\ [\![\texttt{z}]\!])) = (cw{\downarrow}1); \\ \Psi\ (cw{\downarrow}2)\ copy, \\ \qquad \text{otherwise} \end{cases}$
$)\ cw\ cw)\ (\pi_{new}, (\textit{New-w}\ \pi_{new}))$
**where**
$\pi_{new} = (\textit{Pat-fun}\ (\rho_2\ [\![\texttt{z}]\!])) \sqcup (\textit{Mask}\ [\![\texttt{p:z}]\!]\ ((\textit{Pat-fun}\ (\rho\ [\![\texttt{F}]\!]))\ \alpha{\cdot}\pi)\ [\![\texttt{z}]\!])$
$\textit{Mask} = \lambda\ [\![exp]\!]\ \pi\ [\![lv]\!]\ .\ (\textit{Surround}\ [\![exp]\!]\ (\textit{Select-part-of-}\pi\ [\![exp]\!]))$
**where**
$\textit{Select-part-of-}\pi\ =$
$\lambda\Psi.\ \lambda\ [\![exp]\!]\ .$
$\begin{array}{rcl} [\![exp]\!] & = & [\![lv]\!] \longrightarrow \pi, \\ ([\![exp]\!]{\downarrow}1) & = & [\![\texttt{head}]\!] \longrightarrow (\lambda\ \alpha{\cdot}\pi.\ \pi{\downarrow}1)\ (\Psi\ ([\![exp]\!]{\downarrow}1)), \\ & & \qquad\qquad (\lambda\ \alpha{\cdot}\pi.\ \pi{\downarrow}2)\ (\Psi\ ([\![exp]\!]{\downarrow}1)), \end{array}$
$\textit{Surround}\ =$
$\lambda\Psi.\ \lambda\ [\![exp]\!]\ core.$
$\begin{array}{rcl} [\![exp]\!] & = & [\![lv]\!] \longrightarrow core, \\ ([\![exp]\!]{\downarrow}1) & = & [\![\texttt{head}]\!] \longrightarrow \langle(\Psi\ ([\![exp]\!]{\downarrow}2)\ core), \bot\rangle, \\ & & \qquad\qquad \langle\bot, (\Psi\ ([\![exp]\!]{\downarrow}2)\ core)\rangle. \end{array}$

### 4.3.2: Example

An application of [[Rev]] is compiled with the strictness pattern $fix\,\lambda\pi.\langle \$\bot\ ,\ \pi\rangle$ as follows:

```
[[(fix:[Rev λz.
   if:<nil?:head:z
       head:tail:z
       Rev:<tail:head:z .
         <<head:head:z . head:tail:z> . <>>>>)
 <<a . <b . <c>>> . <<> . <>>>]].
```

$[\![\mathtt{z}]\!] \bullet C\,[\![\mathtt{nil?:head:z}]\!]\ \$\bot\ \rho_{init}\ 2 =$

$\$\langle \$\bot\ ,\ \bot\rangle.$

$([\![\mathtt{z}]\!] \bullet C\,[\![\mathtt{nil?:head:z}]\!]\ \$\bot\ \rho_{init}\ 2) \sqcup$

$([\![\mathtt{z}]\!] \bullet C\,[\![\mathtt{head:tail:z}]\!]\ fix\,\lambda\pi.\langle \$\bot\ ,\ \pi\rangle\ \rho_1\ 2) =$

$\$\langle \$\bot\ ,\ \bot\rangle \sqcup \langle \bot\ ,\ \langle fix\,\lambda\pi.\langle \$\bot\ ,\ \pi\rangle\ ,\ \bot\rangle\rangle \sqcap \pi_0$

$= \$\langle \$\bot\ ,\ \langle fix\,\lambda\pi.\langle \$\bot\ ,\ \pi\rangle\ ,\ \bot\rangle\rangle.$

For the rest of this example, $revsyn = ((Pat\text{-}fun\ (\rho\ [\![\mathtt{Rev}]\!]))\ fix\,\lambda\pi.\langle \$\bot\ ,\ \pi\rangle).$

$(Mask\ [\![\mathtt{head:z}]\!]\ revsyn\ [\![\mathtt{z}]\!]) = \langle\ revsyn{\downarrow}1\ ,\ \bot\rangle.$

At this point, the first weakening of the composite pattern is tested:

```
[[<tail:head:z . <<head:head:z.head:tail:z> . <>>>]] •
```

$C\,[\![\mathtt{Rev:<tail:head:z\ .\ <<head:head:z.head:tail:z>\ .\ <>>>}]\!]$

$fix\,\lambda\pi.\langle \$\bot\ ,\ \pi\rangle\ \rho_1\ 2 =$

$\$\langle revsyn{\downarrow}1\ ,\ \langle fix\,\lambda\pi.\langle \$\bot\ ,\ \pi\rangle\ ,\ \bot\rangle\rangle.$

After compilation of [[tail:head:z]],

```
[[z]] • C [[Rev:<tail:head:z.<<head:head:z.head:tail:z>.<>>>]]
```

$fix\,\lambda\pi.\langle \$\bot\ ,\ \pi\rangle\ \rho_1\ 2 =$

$\langle\langle \bot\ ,\ revsyn{\downarrow}1\rangle\ ,\ \bot\rangle.$

After compilation of [[head:head:z]],

$[\![\texttt{z}]\!] \bullet C\,[\![\texttt{Rev:<tail:head:z . <<head:head:z . head:tail:z>.<>>>}]\!]$

$fix\,\lambda\pi.\langle\$\bot\ ,\ \pi\rangle\ \rho_1\ 2 =$

$\$\langle\$\langle\$\bot\ ,\ revsyn{\downarrow}1\rangle\ ,\ \bot\rangle.$

After compilation of $[\![\texttt{head:tail:z}]\!]$,

$[\![\texttt{z}]\!] \bullet C\,[\![\texttt{Rev:<tail:head:z . <<head:head:z . head:tail:z> . <>>>}]\!]$

$fix\,\lambda\pi.\langle\$\bot\ ,\ \pi\rangle\ \rho_1\ 2 =$

$\$\langle\$\langle\$\bot\ ,\ revsyn{\downarrow}1\rangle\ ,\ \langle fix\,\lambda\pi.\langle\$\bot\ ,\ \pi\rangle\ ,\ \bot\rangle\rangle.$

Since (6)

$revsyn{\downarrow}1 = \$\langle\$\bot\ ,\ revsyn{\downarrow}1\rangle,$

$[\![\texttt{z}]\!] \bullet C\,[\![\texttt{Rev:<tail:head:z . <<head:head:z . head:tail:z> . <>>>}]\!]$

$fix\,\lambda\pi.\langle\$\bot\ ,\ \pi\rangle\ \rho_1\ 2 =$

$\$\langle revsyn{\downarrow}1\ ,\ \langle fix\,\lambda\pi.\langle\$\bot\ ,\ \pi\rangle\ ,\ \bot\rangle\rangle.$

Thus

$[\![\texttt{<tail:head:z . <<head:head:z . head:tail:z> . <>>>}]\!]$

$\bullet C\,[\![\texttt{Rev:<tail:head:z . <<head:head:z . head:tail:z> . <>>>}]\!]$

$fix\,\lambda\pi.\langle\$\bot\ ,\ \pi\rangle\ \rho_1\ 2 =$

$[\![\texttt{z}]\!] \bullet C\,[\![\texttt{Rev:<tail:head:z . <<head:head:z . head:tail:z> . <>>>}]\!]$

$fix\,\lambda\pi.\langle\$\bot\ ,\ \pi\rangle\ \rho_1\ 2.$

(6) may be written as $fix\ \lambda\pi.\$\langle\$\bot\ ,\ \pi\rangle.$

$$fix\ \lambda\pi.\$\langle\$\bot\ ,\ \pi\rangle \sqcap \$fix\,\lambda\pi.\langle\$\pi\ ,\ \pi\rangle = \$fix\,\lambda\pi.\langle\$\bot\ ,\ \pi\rangle.$$

Thus

$[\![\texttt{z}]\!] \bullet C\,[\![\texttt{Rev:<<a . <b . <c>>> . <<> . <>>>}]\!]$

$fix\,\lambda\pi.\langle\$\bot\ ,\ \pi\rangle\ \rho_{init}\ \iota =$

$\langle\$fix\,\lambda\pi.\langle\$\bot\ ,\ \pi\rangle\ ,\ \langle fix\,\lambda\pi.\langle\$\bot\ ,\ \pi\rangle\ ,\ \bot\rangle\rangle$

After compilation, the result is

```
⟦(fix:[Rev|fix λπ.⟨$⊥ , π⟩
  λz.
    if:<$nil?:head:z
          head:tail:z
          Rev|fix λπ.⟨$⊥ , π⟩:<$tail:head:z .
            <<$head:head:z . head:tail:z> . $<>>>>)
<$<$a . <$b . <$c>>>  . <$<> . $<>>>⟧.
```

# Chapter 5: Compiling higher order functions

So far, the compiler presented here has been designed to improve first order, list oriented programs. This chapter presents an additional compiler $C''$ developed from $C$, which is an extension of the same approach to higher order functions. This extension is more easily understood if the equations presented in Chapter 2 are first re-structured, so that the additional mechanisms needed can be seen to develop from these equations in a natural way. The restructured compiler, $C'$, which does *not* compile higher order functions is further transformed into $C''$. We then discuss some restrictions upon the source expressions that can be compiled, and relate $C'$ to $C''$, with some examples.

The techniques discussed here have been implemented, but no proofs of correctness are provided.

## Section 5.1: From $C$ to $C'$

Very few changes are required. The essential point is that the syntax specifying the source expressions in $C$ is no longer appropriate, as lambda expressions were constrained in Chapter 2 to appear only as part of an application. In order to discuss an extension of $C$ to higher order functions, it is useful to first construct a syntax that can be used by both $C'$ and $C''$, so that the equations are directly comparable. However, the constraint is still present, in the sense that no guarantees are made about $C''$'s behavior on programs which return lambda expressions as values.

Now that lambda expressions are no longer syntactically related to application, it is more elegant to have some explicit mechanism for transmitting the pattern synthesized by the compilation of a lambda body back to the compilation of the application of that expression. This is done by expanding the number of arguments given to $C$ to five; the additional argument follows the inherited pattern and serves as the synthesized pattern produced by the compilation of each expression.

### 5.1.1: New Daisy syntax and compiler domains

| | |
|---|---|
| e ::= expr \| $expr | *constants* |
| expr ::= const \| | *nil* |
| []\| | *lists* |
| ⟨ exprs ⟩\| | *primitives with 2 arguments* |
| prim:⟨e e⟩ \| | *head application* |
| head:e \| | *tail application* |
| tail:e \| | *conditional application* |
| if:⟨e e e⟩ \| | *lambda expressions* |
| λ id. e \| | *recursive function and data definitions* |
| fix :[id e] \| | *function application* |
| e:e \| | *identifiers* |
| id \| | *infinite loop* |
| bottom | |

exprs ::= e exprs | e . e | *empty*

The compiler domains are virtually unchanged. Instead of propagating strictness patterns only through its second argument and recovering synthesized patterns from the compiler environment, the compiler now distinguishes between inherited and synthesized patterns by propagating inherited patterns through its second argument and synthesized patterns through its third argument.

$C'$ : $D \longrightarrow D$; (compiler)

$D =$ $EXP \times \mathbf{P} \times \mathbf{P} \times ENV \times INT$ ; (compilation data)

$\pi$ : $\mathbf{P} =$ $\$\mathbf{P} + (\mathbf{P} \times \mathbf{P})$; (inherited strictness patterns)

$\sigma$ : $\mathbf{P} =$ $\$\mathbf{P} + (\mathbf{P} \times \mathbf{P})$; (synthesized strictness patterns)

### 5.1.2: Equations for $C'$

The first seven equations are essentially unchanged, except for the addition of a new argument, $\sigma$, that follows $\alpha{\cdot}\pi$.

$$C'\ [\![\texttt{const}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota = [\![\$\texttt{const}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota. \qquad (C'\ 1)$$

$$C'\ [\![\texttt{e}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota, \textbf{where}\ (\$ \notin \alpha{\cdot}\pi) = \qquad (C'\ 2)$$

$$\begin{cases} [\![\texttt{e}[\,[\![\texttt{fix:[id exp]}]\!] / [\![\texttt{e}']\!]\,]]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota \\ \textbf{where}\ [\![\texttt{fix:[id exp]}]\!] = \\ \quad (\textit{Binding}\ (\rho\ [\![\texttt{e}']\!])) \\ \qquad \text{if } \exists\ [\![\texttt{e}']\!] \in [\![\texttt{e}]\!]\ \textit{such that}\ [\![\texttt{e}']\!] \in \textit{ID} \\ \qquad \&\ (\textit{Binding-type}\ (\rho\ [\![\texttt{e}']\!])) = \textbf{fix}\ , \\ \\ [\![\texttt{e}[\,[\![\texttt{(fix:[f }\lambda\texttt{id. body]):exp}]\!] / [\![\texttt{e}']\!]\,]]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota \\ \textbf{where}\ [\![\texttt{(fix:[f }\lambda\texttt{id. body])}]\!] = \\ \quad (\textit{Binding}\ (\rho\ [\![\texttt{f}]\!])) \\ \qquad \text{if } \exists\ [\![\texttt{e}']\!] \in [\![\texttt{e}]\!]\ \textit{such that}\ [\![\texttt{e}']\!] = [\![\texttt{f:exp}]\!] \\ \qquad \&\ (\textit{Binding-type}\ (\rho\ [\![\texttt{f}]\!])) = \textbf{fix}\ , \\ \\ [\![\texttt{e}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota \\ \qquad \text{otherwise.} \end{cases}$$

$$C'\ [\![\texttt{head:e}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota = [\![\texttt{head:e}_1]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho_1\ \iota \qquad (C'\ 3)$$
$$\textbf{where}\ [\![\texttt{e}_1]\!]\ \alpha_1{\cdot}\pi_1\ \sigma_1\ \rho_1\ \iota_1 = C'\ [\![\texttt{e}]\!]\ \alpha{\cdot}\langle\alpha{\cdot}\pi,\ \bot\rangle\ \sigma\ \rho\ \iota.$$

$$C'\,[\![\texttt{tail:e}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota = [\![\texttt{tail:e}_1]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho_1\ \iota$$
$$\textbf{where}\ [\![\texttt{e}_1]\!]\ \alpha_1{\cdot}\pi_1\ \sigma_1\ \rho_1\ \iota_1 = C'\,[\![\texttt{e}]\!]\ \alpha{\cdot}\langle\perp,\ \alpha{\cdot}\pi\rangle\ \sigma\ \rho\ \iota.$$

($C'$ 4)

$$C'\,[\![\texttt{<e1 . e2>}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota =$$
$$\begin{cases} C'\,[\![\texttt{<e1 . e2>}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota, \textbf{where}\ (\$ \notin \alpha{\cdot}\pi) & \text{if } (\$ \notin \pi); \\ [\![\texttt{<}\alpha_1{\cdot}\texttt{e1}_1\texttt{.}\ \ \alpha_2{\cdot}\texttt{e2}_2\texttt{>}]\!]\ \alpha{\cdot}\pi\ \sigma_2\ \rho_2\ \iota, & \text{otherwise;} \end{cases}$$
$$\textbf{where}$$
$$\alpha_1{\cdot}\pi_1 = (\pi{\downarrow}1)$$
$$\alpha_2{\cdot}\pi_2 = (\pi{\downarrow}2)$$
$$[\![\texttt{e1}_1]\!]\ \alpha_1{\cdot}\pi_1\ \sigma_1\ \rho_1\ \iota_1 = C'\,[\![\texttt{e1}]\!]\,(\pi{\downarrow}1)\ \sigma\ \rho\ \iota;$$
$$[\![\texttt{e2}_2]\!]\ \alpha_2{\cdot}\pi_2\ \sigma_2\ \rho_2\ \iota_2 = C'\,[\![\texttt{e2}]\!]\,(\pi{\downarrow}2)\ \sigma\ \rho_1\ \iota.$$

($C'$ 5)

$$C'\,[\![\texttt{prim:<e1 e2>}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota = [\![\texttt{prim:e}_1]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho_1\ \iota$$
$$\textbf{where}$$
$$[\![\texttt{e}_1]\!]\ \alpha_1{\cdot}\pi_1\ \sigma_1\ \rho_1\ \iota_1 = C'\,[\![\texttt{<e1 e2>}]\!]\ \langle\$\perp\ ,\ \$\langle\$\perp\ ,\ \perp\rangle\rangle\sigma\ \rho\ \iota.$$

($C'$ 6)

$(C'\ 7)$

$$C'\ [\![\texttt{if:<e1 e2 e3>}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota$$
$$= [\![\texttt{if:<\$e1}_1\ \texttt{e2}_2\ \texttt{e3}_3\texttt{>}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho_4\ \iota$$

**where**

$$[\![\texttt{e1}_1]\!]\ \alpha_1{\cdot}\pi_1\ \sigma_1\ \rho_1\ \iota_1 = C'\ [\![\texttt{e1}]\!]\ \$\bot\ \sigma\ \rho\ \iota;$$
$$[\![\texttt{e2}_2]\!]\ \alpha_2{\cdot}\pi_2\ \sigma_2\ \rho_2\ \iota_2 = C'\ [\![\texttt{e2}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho_1\ \iota;$$
$$[\![\texttt{e3}_3]\!]\ \alpha_3{\cdot}\pi_3\ \sigma_3\ \rho_3\ \iota_3 = C'\ [\![\texttt{e3}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho_1\ \iota;$$

$$\rho_4 = \lambda i.\begin{cases}\langle [\![\texttt{binding}_2]\!], (pa_2 \sqcap pa_3), 0, b\text{-}type_2\rangle \\ \qquad \text{if } b\text{-}type_2 = \textbf{lambda}; \\ \langle [\![\texttt{binding}_2]\!], pa_4, v\text{-}count_2 + v\text{-}count_3, b\text{-}type_2\rangle \\ \qquad \text{if } b\text{-}type_2 = \textbf{fix}\ ;\end{cases}$$

**where**

$$\langle [\![\texttt{binding}_2]\!], pa_2, v\text{-}count_2, b\text{-}type_2\rangle = \rho_2\ i$$
$$\langle [\![\texttt{binding}_3]\!], pa_3, v\text{-}count_3, b\text{-}type_3\rangle = \rho_3\ i$$
$$pa_4 = \lambda pat.\begin{cases}(pa_3\ pat) & \text{if } (pa_2\ pat) = unbound; \\ (pa_2\ pat) & \text{otherwise.}\end{cases}$$

$(C'\ 8)$

$$C'\ [\![\lambda\texttt{id. body}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota =$$
$$[\![\lambda\texttt{id. body}_1]\!]\ \alpha{\cdot}\pi\ (Pat\text{-}fun\ (\rho_1\ [\![\texttt{id}]\!]))\ \rho_3\ \iota$$

**where**

$$[\![\texttt{body}_1]\!]\ \alpha_1{\cdot}\pi_1\ \sigma_1\ \rho_1\ \iota_1 = C'\ [\![\texttt{body}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho_2\ \iota$$
$$\rho_2 = \lambda i.\begin{cases}\langle [\![[]]\!], \bot, 0, \textbf{lambda}\rangle, & \text{if } i = [\![\texttt{id}]\!]; \\ \rho\ i, & \text{otherwise;}\end{cases}$$
$$\rho_3 = \lambda i.\begin{cases}\rho\ i, & \text{if } i = [\![\texttt{id}]\!]; \\ \rho_1\ i, & \text{otherwise.}\end{cases}$$

The compilation of a lambda expression was previously embedded in the compilation of a *fix* expression. Here, the compilation of the function body generates a synthesized pattern which represents the patterns accumulated during compilation of all instances of $[\![\texttt{id}]\!]$. This synthesized pattern is passed back to the compilation of an application of the lambda expression.

$$C'\,[\![\texttt{fix:[id e]}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota = [\![\texttt{fix:[id}|\alpha{\cdot}\pi\ \texttt{e}_1\texttt{]}]\!]\ \alpha{\cdot}\pi\ \sigma_1\ \rho_3\ \iota \qquad (C'\ 9)$$

**where**

$$[\![\texttt{e}_1]\!]\ \alpha_1{\cdot}\pi_1\ \sigma_1\ \rho_1\ \iota_1 = C'\,[\![\texttt{e}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho_2\ \iota;$$

$$\rho_2 = \lambda i.\begin{cases} \langle\, [\![\texttt{fix:[id e]}]\!], pa_2, 1, \textbf{fix}\,\rangle, \\ \qquad \text{if } i = [\![\texttt{id}]\!]; \\ \rho\, i, \text{otherwise;} \end{cases}$$

$$pa_2 = \lambda pat.\begin{cases} \sigma_1 & \text{if } pat = \alpha{\cdot}\pi; \\ unbound & \text{otherwise;} \end{cases}$$

$$\rho_3 = \lambda i.\begin{cases} \rho\, i, & \text{if } i = [\![\texttt{id}]\!]; \\ \rho_1\, i, & \text{otherwise.} \end{cases}$$

Data recursion and recursive functions are now compiled using the same equation. Note that $\sigma_1$ is actually *rec-p* (see Chapter 2), which is now passed back explicitly as a synthesized pattern from the analysis of the lambda body. If $[\![\texttt{e}]\!]$ is not a lambda expression, then *rec-p* will not be propagated as a synthesized pattern and so can safely be associated with $\alpha{\cdot}\pi$ in $pa_2$.

$$(C'\ 10)$$

$$C'\,[\![\texttt{f:e}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota = [\![\texttt{tf:e}_2]\!]\ \alpha{\cdot}\pi\ \sigma_2\ \rho_2\ \iota$$

**where**

$$[\![\texttt{f}_1]\!]\ \alpha_1{\cdot}\pi_1\ \sigma_1\ \rho_1\ \iota_1 = \begin{cases} C'\,[\![\texttt{f}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota & \text{if } [\![\texttt{f}]\!] \in V; \\ C'\,[\![\texttt{sf}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota & \\ \qquad \textbf{where } [\![\texttt{(sf)}]\!] = [\![\texttt{f}]\!] & \text{otherwise;} \end{cases}$$

$$[\![\texttt{e}_2]\!]\ \alpha_2{\cdot}\pi_2\ \sigma_2\ \rho_2\ \iota_2 = C'\,[\![\texttt{e}]\!]\ \sigma_1\ \sigma\ \rho_1\ \iota$$

$$[\![\texttt{tf}]\!] = \begin{cases} [\![\texttt{f}_1]\!] & \text{if } [\![\texttt{f}_1]\!] \in V; \\ [\![\texttt{(f}_1\texttt{)}]\!] & \text{otherwise.} \end{cases}$$

Function application is straightforward, except that the function may sometimes be a parenthesized expression. In addition, the compilation of a function may produce an expression that must be parenthesized in order to preserve precedence.

($C'$ 11)

$C'\ [\![\texttt{id}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota =$

$$\begin{cases} \textbf{Variable} & \text{if } b\text{-}type = \textbf{lambda}; \\ \textbf{Reached-Limit} & \text{if } (pa\ \alpha{\cdot}\pi) \neq unbound\ \&\ v\text{-}count \geq \iota; \\ \textbf{Compile-Binding} & \text{if } (pa\ \alpha{\cdot}\pi) \neq unbound\ \&\ v\text{-}count < \iota; \\ \textbf{Mark-With-Pattern} & \text{otherwise;} \end{cases}$$

**where** $\langle\ [\![\texttt{binding}]\!]\ , pa, v\text{-}count, b\text{-}type\rangle = \rho\ [\![\texttt{id}]\!]\ ;$

**Variable is**
$[\![\texttt{id}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho_1\ \iota$
**where**

$$\rho_1 = \lambda i.\begin{cases} \langle\ [\![\texttt{binding}]\!]\ ,\ (\alpha{\cdot}\pi \sqcup pa) \sqcap \pi_0, v\text{-}count, b\text{-}type\rangle, \\ \qquad \text{if } i = [\![\texttt{id}]\!]\ ; \\ \rho\ i, \text{otherwise;} \end{cases}$$

**Reached-Limit is** $[\![\texttt{binding}]\!]\ \alpha{\cdot}\pi\ \perp\ \rho\ \iota;$

**Compile-Binding is**
$[\![\texttt{fix:[id|}\alpha{\cdot}\pi\ \texttt{e}_1\texttt{]}]\!]\ \alpha{\cdot}\pi\ \sigma_1\ \rho_3\ \iota$
**where**
$[\![\texttt{e}_1]\!]\ \alpha_1{\cdot}\pi_1\ \sigma_1\ \rho_1\ \iota_1 = C\ [\![\texttt{e}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho_1\ \iota;$

$$\rho_2 = \lambda i.\begin{cases} \langle\ [\![\texttt{fix:[id e]}]\!]\ , pa_2, v\text{-}count + 1, \textbf{fix}\ \rangle, \\ \qquad \text{if } i = [\![\texttt{id}]\!]\ ; \\ \rho\ i, \text{otherwise;} \end{cases}$$

$$pa_2 = \lambda pat.\begin{cases} \sigma_1 & \text{if } pat = \alpha{\cdot}\pi; \\ (pa\ pat) & \text{otherwise;} \end{cases}$$

$$\rho_3 = \lambda i.\begin{cases} \rho\ i, & \text{if } i = [\![\texttt{id}]\!]\ ; \\ \rho_1\ i, & \text{otherwise.} \end{cases}$$

**Mark-With-Pattern is** $[\![\texttt{id}|\alpha{\cdot}\pi]\!]\ \alpha{\cdot}\pi\ (pa\ \alpha{\cdot}\pi)\ \rho\ \iota.$

The compilation of an identifier now includes identifiers as functions as well as identifiers representing data recursions. Note that synthesized patterns are propagated by both **Reached-Limit** and **Mark-With-Pattern**, because the identifier may represent a function whose argument must be compiled with a synthesized pattern.

## Section 5.2: Restrictions upon source expressions

Before discussing the extension of $C'$ into $C''$, it is necessary to restrict the source expressions that may be compiled in three ways. Briefly, these restrictions require that

— functions are defined at compile time

— source expressions are correctly typed

— functions are not returned as values *by the entire program*

These constraints are discussed more fully in the following paragraphs.

### 5.2.1: Functions must be defined at compile time.

It is assumed that all functions are defined at compile time, as the compiler must be able to identify their definitions in order to annotate them.

### 5.2.2: Expressions must be correctly typed.

We wish to use a fast, easily understood algorithm, and so would like to continue to construct a recursive descent compiler. This means that when compiling the expression

$[\![$(λf. (head:f):<h . <>>):<λg. add:<head:g head:g> . <>>$]\!]$,

the compiler should be able to determine the pattern that the appropriate version of $[\![$f$]\!]$ synthesizes so that it can analyze $[\![$h$]\!]$ before leaving the scope in which $[\![$f$]\!]$ is defined. This can be done very simply by passing the expression to which $[\![$f$]\!]$ is bound in to the compilation of $[\![$(head:f):<h . <>>$]\!]$, so that it is available when needed. Since versions are created, and since a lazy version must be retained in case there is no better option, the compiled binding of $[\![$f$]\!]$ is substituted for the identifier $[\![$f$]\!]$, leaving the original binding of $[\![$f$]\!]$ to be called if an application of $[\![$f$]\!]$ is compiled as if it were lazy. Thus, the compiler must be able to recognize a formal (the bound variable of a lambda expression) representing a function or a structure containing a function, because such a formal is compiled by copying the expression bound to it in place of the instance of the formal; this is distinct from the treatment given a formal that represents any other value. A type checker

is expected to provide this information, and programs in the restricted Daisy language are now presumed to be polymorphically typed [5].

**5.2.3: Functions cannot be returned as values by the entire program.**

This restriction is derived from the fact that the compiler has no way of ensuring that all uses of a returned function will occur in an application which inherits the printer pattern. The user could of course apply the resulting function without any guarantee of its safety.

One further assumption made here is that the programmer uses *fix*, not the Y combinator, when writing loops.

## Section 5.3: An extension of $C'$ to $C''$

Briefly, the changes introduced in these equations are that

— $\sigma$ is now a tree of synthesized patterns rather than just a single pattern

— A new compiler argument, $\zeta$, is added to provide a stack of expressions, each of which is the argument of a function application. When necessary, an argument is entered into the compiler environment so that it may replace instances of calls to the formal or part of the formal when required.

These changes are justified in sections 5.3.2 and 5.3.3. In addition, the compiler environment is extended to include one more item per entry. This item is a tag, derived from information produced by a type checker, which permits the compiler to categorize the bound expression's type. The type information required is crude, and is interpreted by the compiler as follows: if the bound expression's value is neither a function nor a structure containing a function, then the compiler assumes it is of type **var**, otherwise, the compiler assumes the bound expression's value is of type **function**.

**5.3.1: New compiler domains**

$C''$ :

$$D \longrightarrow D;$$

*(compiler)*

$$D = EXP \times \mathbf{P} \times PT \times ENV' \times INT \times EXP^*;$$

(compilation data)

$$\pi : \mathbf{P} = \$\mathbf{P} + (\mathbf{P} \times \mathbf{P});$$

(inherited strictness patterns)

$$\sigma : PT = (\mathbf{P} \times []) + (PT \times PT);$$

(tree of synthesized strictness patterns)

$$\rho : ENV' = V \longrightarrow (EXP \times PF \times INT \times BTAG \times TTAG) + unbound;$$

(compiler environment)

$$\iota : INT;$$

(resource)

$$\zeta : EXP^*$$

(expression stack)

$$\nu : V = ID + (ID \times \mathbf{P});$$

(version identifiers)

$$pa : PF = (\mathbf{P} \longrightarrow (\mathbf{P}_{\pi_0} + unbound)) + \mathbf{P}_{\pi_0};$$

(inherited and synthesized pattern entries in environment)

$$BTAG = \textbf{lambda} + \textbf{fix}$$

(binding tags in environment)

$$TTAG = \textbf{var} + \textbf{function}$$

(type information produced by type checker)

### 5.3.2: Synthesized patterns are passed around in a stack.

The compilation of higher order functions requires a mechanism for returning more than one synthesized pattern from the compilation of an expression. For example, it would be useful to be able to compile

$$[\![((\lambda \texttt{a}.\ \lambda \texttt{b}.\ \texttt{<head:a . head:b>}):\texttt{<c . <>>}):\texttt{<d . <>>}]\!] \quad (1)$$

$C''$'s equation handling lambda expressions, ($C'$ 8), received a single synthesized pattern from the compilation of the function and passed this pattern on

to the compilation of the argument, which is sufficient for the compilation of expressions in a first-order language. In fact, it was possible to take advantage of this and compress inherited and synthesized patterns into one compiler variable, $\alpha{\cdot}\pi$, by viewing a synthesized pattern as just another name for a pattern inherited by a bound variable at the moment when compilation of the binding lambda expression is completed. We extend this approach to compile expressions like (1) by distinguishing between inherited and synthesized patterns as follows:

— Each compilation now inherits a single pattern, $\alpha{\cdot}\pi$, which is used as in Chapter 2.

— A new variable, $\sigma$, represents a tree of synthesized patterns which is returned by each compilation.

The compilation of a lambda expression now adds the synthesized pattern to the tree returned by the compilation of the lambda body. Compilation of a function application selects the top pattern on the tree and passes it to the compilation of the argument. (Much of this discussion makes it appear that this tree is actually a stack— it is, until the equation for **cons** has to combine stacks, so that the equations for **head** and **tail** can take them apart. For this reason, it is referred to it as a tree.)

Many of the equations simply return the pattern tree passed to them unless they have a specific use for it. However, there are exceptions, such as the equations for **head**, **tail**, **cons**, **if**, **fix** and **id**; these are discussed when the equations are presented.

### 5.3.3: Application arguments are also passed around in a stack.

Consider the following expression;

$[\![$`((λh. λf. <h . (head:f):<b . <>>>):<2 . <>>):<g . <>>`$]\!]$ (2)

Unless the compiler can determine what pattern should be synthesized for $[\![$`f`$]\!]$, it cannot analyze $[\![$`b`$]\!]$. If it does not know what expression $[\![$`f`$]\!]$ will eventually be bound to when the application $[\![$`(head:f):<b>`$]\!]$ is to be compiled, then it

must halt the compilation of ⟦<h . (head:f):<b>>⟧ and search the surrounding expression for ⟦f⟧'s binding.

It seems simpler to carry the binding of ⟦f⟧ into the compilation of the expression in which ⟦f⟧ may be applied, so that the compiler may produce an appropriate version at the point at which it discovers an application of ⟦f⟧. In (2), it is easy to see that the compiler requires a stack of application arguments, rather than a single argument. Here, ⟦h⟧ will eventually be bound to the value of ⟦<2 . <>>⟧, and ⟦f⟧ to ⟦<g . <>>⟧.

A stack of application arguments is represented by $\zeta$, which grows each time an application is compiled. It shrinks during the compilation of a lambda expression, when the top expression on the stack is entered into the compile-time environment so that it is available for compilation whenever an application of its bound variable is compiled.

**Section 5.3.4: Compiler equations for $C''$**

The first two equations contain the new compiler variables $\sigma$ and $\zeta$, but are otherwise unchanged:

$$C''\,⟦\texttt{const}⟧\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota\ \zeta = ⟦\$\texttt{const}⟧\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota\ \zeta. \qquad (C''\ 1)$$

$(C''\ 2)$

$$C''\ [\![\mathtt{e}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota\ \zeta, \textbf{where}\ (\$ \notin \alpha{\cdot}\pi) =$$

$$\begin{cases} [\![\mathtt{e}[\,[\![\mathtt{fix:[id\ exp]}]\!]\,/\,[\![\mathtt{e'}]\!]\,]]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota\ \zeta \\ \textbf{where}\ [\![\mathtt{fix:[id\ exp]}]\!] = \\ \quad (\textit{Binding}\ \ (\rho\ [\![\mathtt{e'}]\!])) \\ \qquad\qquad \text{if } \exists\ [\![\mathtt{e'}]\!] \in [\![\mathtt{e}]\!]\ \textit{such that}\ [\![\mathtt{e'}]\!] \in \mathit{ID} \\ \qquad\qquad \&\ (\textit{Binding-type}\ (\rho\ [\![\mathtt{e'}]\!])) = \textbf{fix}\ , \\ \\ [\![\mathtt{e}[\,[\![\mathtt{(fix:[f\ \lambda id.\ body])\!:exp}]\!]\,/\,[\![\mathtt{e'}]\!]\,]]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota\ \zeta \\ \textbf{where}\ [\![\mathtt{(fix:[f\ \lambda id.\ body])}]\!] = \\ \quad (\textit{Binding}\ \ (\rho\ [\![\mathtt{f}]\!])) \\ \qquad\qquad \text{if } \exists\ [\![\mathtt{e'}]\!] \in [\![\mathtt{e}]\!]\ \textit{such that}\ [\![\mathtt{e'}]\!] = [\![\mathtt{f:exp}]\!] \\ \qquad\qquad \&\ (\textit{Binding-type}\ (\rho\ [\![\mathtt{f}]\!])) = \textbf{fix}\ , \\ \\ [\![\mathtt{e}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota\ \zeta \\ \qquad\qquad \text{otherwise.} \end{cases}$$

$(C''\ 3)$

$$C''\ [\![\mathtt{head:e}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota\ \zeta = [\![\mathtt{head:e_1}]\!]\ \alpha{\cdot}\pi\ (\sigma_1 = () \longrightarrow (), (\sigma_1{\downarrow}1))\ \rho_1\ \iota\ \zeta_1$$
$$\textbf{where}\ [\![\mathtt{e_1}]\!]\ \alpha_1{\cdot}\pi_1\ \sigma_1\ \rho_1\ \iota_1\ \zeta_1 = C''\ [\![\mathtt{e}]\!]\ \alpha{\cdot}\langle\alpha{\cdot}\pi,\ \bot\rangle\ \sigma\ \rho\ \iota\ \zeta.$$

$(C''\ 4)$

$$C''\ [\![\mathtt{tail:e}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota\ \zeta = [\![\mathtt{tail:e_1}]\!]\ \alpha{\cdot}\pi\ (\sigma_1 = () \longrightarrow (), (\sigma_1{\downarrow}2))\ \rho_1\ \iota\ \zeta_1$$
$$\textbf{where}\ [\![\mathtt{e_1}]\!]\ \alpha_1{\cdot}\pi_1\ \sigma_1\ \rho_1\ \iota_1\ \zeta_1 = C''\ [\![\mathtt{e}]\!]\ \alpha{\cdot}\langle\bot,\ \alpha{\cdot}\pi\rangle\ \sigma\ \rho\ \iota\ \zeta.$$

$$C''\ [\![\texttt{<e1 . e2>}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota\ \zeta = \begin{cases} C''\ [\![\texttt{<e1 . e2>}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota\ \zeta, \textbf{ where } (\$ \notin \alpha{\cdot}\pi) & \text{if } (\$ \notin \pi); \\ [\![\texttt{<}\alpha_1{\cdot}\texttt{e1}_1\texttt{ . }\ \alpha_2{\cdot}\texttt{e2}_2\texttt{>}]\!]\ \alpha{\cdot}\pi\ \langle\sigma_1, \sigma_2\rangle\ \rho_2\ \iota_2\ \zeta_2, & \text{otherwise;} \end{cases}$$

**where**

$\alpha_1{\cdot}\pi_1 = (\pi{\downarrow}1)$

$\alpha_2{\cdot}\pi_2 = (\pi{\downarrow}2)$

$[\![\texttt{e1}_1]\!]\ \alpha_1{\cdot}\pi_1\ \sigma_1\ \rho_1\ \iota_1\ \zeta_1 = C''\ [\![\texttt{e1}]\!]\ (\pi{\downarrow}1)\ \sigma\ \rho\ \iota\ \zeta;$

$[\![\texttt{e2}_2]\!]\ \alpha_2{\cdot}\pi_2\ \sigma_2\ \rho_2\ \iota_2\ \zeta_2 = C''\ [\![\texttt{e2}]\!]\ (\pi{\downarrow}2)\ \sigma\ \rho_1\ \iota\ \zeta.$

($C''$ 5)

The equations for **head, tail** and **cons** are altered to accomodate the compilation of lists of functions. The trees returned by the compilation of **cons** sub-expressions are paired; these pairs are de-structured by the equations for **head** and **tail**.

$$C''\ [\![\texttt{prim:<e1 e2>}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota\ \zeta = [\![\texttt{prim:e}_1]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho_1\ \iota\ \zeta_1$$

**where**

$[\![\texttt{e}_1]\!]\ \alpha_1{\cdot}\pi_1\ \sigma_1\ \rho_1\ \iota_1\ \zeta_1 = C''\ [\![\texttt{<e1 e2>}]\!]\ \langle \$\bot\ ,\ \$\langle \$\bot\ ,\ \bot\rangle\rangle \sigma\ \rho\ \iota\ \zeta.$

($C''$ 6)

$$
\begin{array}{l}
C''\ [\![\texttt{if:<e1 e2 e3>}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota\ \zeta \\
\qquad = [\![\texttt{if:<\$e1}_1\ \texttt{e2}_2\ \texttt{e3}_3\texttt{>}]\!]\ \alpha{\cdot}\pi\ (\mathit{Meet\text{-}elts}\ \sigma_2\ \sigma_3)\ \rho_4\ \iota\ \zeta_3 \\
\textbf{where} \\
[\![\texttt{e1}_1]\!]\ \alpha_1{\cdot}\pi_1\ \sigma_1\ \rho_1\ \iota_1\ \zeta_1 = C''\ [\![\texttt{e1}]\!]\ \$\bot\ \sigma\ \rho\ \iota\ \zeta; \\
[\![\texttt{e2}_2]\!]\ \alpha_2{\cdot}\pi_2\ \sigma_2\ \rho_2\ \iota_2\ \zeta_2 = C''\ [\![\texttt{e2}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho_1\ \iota\ \zeta_1; \\
[\![\texttt{e3}_3]\!]\ \alpha_3{\cdot}\pi_3\ \sigma_3\ \rho_3\ \iota_3\ \zeta_3 = C''\ [\![\texttt{e3}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho_1\ \iota\ \zeta_1; \\
\rho_4 = \lambda i. \begin{cases} \langle\ [\![\texttt{binding}_2]\!]\ ,\ (pa_2 \sqcap pa_3)\ , 0, \mathit{b\text{-}type}_2, \mathit{var\text{-}type}_2 \rangle \\ \qquad\qquad \text{if } \mathit{b\text{-}type}_2 = \textbf{lambda}; \\ \langle\ [\![\texttt{binding}_2]\!]\ , pa_4, \mathit{v\text{-}count}_2 + \mathit{v\text{-}count}_3, \mathit{b\text{-}type}_2, \mathit{var\text{-}type}_2 \rangle \\ \qquad\qquad \text{if } \mathit{b\text{-}type}_2 = \textbf{fix}\ ; \end{cases} \\
\textbf{where} \\
\quad \langle\ [\![\texttt{binding}_2]\!]\ , pa_2, \mathit{v\text{-}count}_2, \mathit{b\text{-}type}_2, \mathit{var\text{-}type}_2 \rangle = \rho_2\ i \\
\quad \langle\ [\![\texttt{binding}_3]\!]\ , pa_3, \mathit{v\text{-}count}_3, \mathit{b\text{-}type}_3, \mathit{var\text{-}type}_3 \rangle = \rho_3\ i \\
\quad pa_4 = \lambda pat. \begin{cases} (pa_3\ pat) & \text{if } (pa_2\ pat) = \mathit{unbound}; \\ (pa_2\ pat) & \text{otherwise}; \end{cases} \\
\mathit{Meet\text{-}elts} = \\
\qquad \mathit{fix}\lambda\Psi.\lambda l^1 l^2. \\
\qquad\quad l^1 = () \longrightarrow (), \\
\qquad\quad l^1{\downarrow}1 \in \mathbf{P} \longrightarrow (\ ((l^1{\downarrow}1) \sqcap (l^2{\downarrow}1)),\ \Psi(l^1{\downarrow}2)(l^2{\downarrow}2)\ ), \\
\qquad\qquad\qquad (\ \Psi(l^1{\downarrow}1)\ (l^2{\downarrow}1),\ \Psi(l^1{\downarrow}2)(l^2{\downarrow}2)\ ).
\end{array}
\qquad (\mathcal{C}''\ 7)
$$

The equation for **if** must be able to compile expressions such as

$$[\![\texttt{if:<p:2 }\lambda\texttt{a. w }\lambda\texttt{b. x>}]\!],$$

and so must return a tree which contains $a \sqcap b$ for all $a \in \sigma^2$ and corresponding $b \in \sigma^3$. Other than this, $(\mathcal{C}''\ 7)$ is equivalent to $(\mathcal{C}'\ 7)$, except that it contains the new compiler variable $\zeta$. Note that $\sigma_2$ and $\sigma_3$ must have the same structure, which means that if the two branches return lists, they must have the same internal tree structure.

$$C''\ [\![\lambda \texttt{id. body}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota\ \langle\ [\![\texttt{e}]\!]\ , \zeta\rangle = \qquad (C''\ 8)$$
$$[\![\lambda \texttt{id. body}_1]\!]\ \alpha{\cdot}\pi\ \langle (\mathit{Pat\text{-}fun}\ (\rho_1\ [\![\texttt{id}]\!]\ )), \sigma_1\rangle\ \rho_3\ \iota\ \zeta_1$$

**where**

$$[\![\texttt{body}_1]\!]\ \alpha_1{\cdot}\pi_1\ \sigma_1\ \rho_1\ \iota_1\ \zeta_1 = C''\ [\![\texttt{body}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho_2\ \iota\ \zeta$$

$$\rho_2 = \lambda i. \begin{cases} \langle\ [\![\texttt{e}]\!]\ , \bot, 0, \textbf{lambda}, (\mathit{Type}\ [\![\texttt{id}]\!]\ )\rangle, & \text{if } i = [\![\texttt{id}]\!]\ ; \\ \rho\ i, & \text{otherwise;} \end{cases}$$

$$\rho_3 = \lambda i. \begin{cases} \rho\ i, & \text{if } i = [\![\texttt{id}]\!]\ ; \\ \rho_1\ i, & \text{otherwise.} \end{cases}$$

Like the corresponding equations of $C$ and $C'$, ($C''$ 8) requires that the compiler analyze the body of the lambda expression in an environment in which a new entry is created for the local variable. This entry now contains the text of the expression whose value will be bound to the local variable at run time. Note that this expression, $[\![\texttt{e}]\!]$, is propagated to the compilation of the lambda expression on the argument stack, $\zeta$. If it is never required, then $[\![\texttt{e}]\!]$ may still be present on the stack when the compiler has finished compiling the application; if this is the case, it is removed.

*Type* maps information provided by the type checker into an element in {**var**, **function**}.

$$C''\ [\![\texttt{fix:[id e]}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota\ \zeta = [\![\texttt{fix:[id}|\alpha{\cdot}\pi\ \texttt{e}_1\texttt{]}]\!]\ \alpha{\cdot}\pi\ \sigma_1\ \rho_3\ \iota\ \zeta_1 \qquad (C''\ 9)$$

**where**

$$[\![\texttt{e}_1]\!]\ \alpha_1{\cdot}\pi_1\ \sigma_1\ \rho_1\ \iota_1\ \zeta_1 = C''\ [\![\texttt{e}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho_2\ \iota\ \zeta;$$

$$\rho_2 = \lambda i. \begin{cases} \langle\ [\![\texttt{fix:[id e]}]\!]\ , pa_2, 1, \textbf{fix}\ , (\mathit{Type}\ [\![\texttt{id}]\!]\ )\rangle, \\ \quad \text{if } i = [\![\texttt{id}]\!]\ ; \\ \rho\ i, \text{otherwise;} \end{cases}$$

$$pa_2 = \lambda pat. \begin{cases} \sigma_1 & \text{if } pat = \alpha{\cdot}\pi; \\ \mathit{unbound} & \text{otherwise;} \end{cases}$$

$$\rho_3 = \lambda i. \begin{cases} \rho\ i, & \text{if } i = [\![\texttt{id}]\!]\ ; \\ \rho_1\ i, & \text{otherwise.} \end{cases}$$

($C''$ 9) is very similar to ($C'$ 9). The only differences are that now the type of $[\![\texttt{id}]\!]$ is entered into the compiler environment, and that the expression stack has been added.

($C''$ 10)

$$
\begin{array}{l}
C\ [\![\texttt{f:e}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota\ \zeta = \\
\quad [\![\texttt{tf:e}_2]\!]\ \alpha{\cdot}\pi\ \sigma_2\ \rho_2\ \iota\ ((\zeta_2{\downarrow}1) = [\![\texttt{e}]\!]\ \longrightarrow\ (\zeta_2{\downarrow}2), \zeta_2) \\
\textbf{where} \\
[\![\texttt{f}_1]\!]\ \pi_1\ \sigma_1\ \rho_1\ \iota_1\ \zeta_1 = \\
\quad \begin{cases} C\ [\![\texttt{f}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota\ \langle [\![\texttt{e}]\!], \zeta \rangle & \text{if } [\![\texttt{f}]\!] \in V; \\ C\ [\![\texttt{sf}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota\ \langle [\![\texttt{e}]\!], \zeta \rangle & \\ \qquad \textbf{where } [\![\texttt{(sf)}]\!] = [\![\texttt{f}]\!] & \text{otherwise;} \end{cases} \\
[\![\texttt{e}_2]\!]\ \pi_2\ \sigma_2\ \rho_2\ \iota_2\ \zeta_2 = C\ [\![\texttt{e}]\!]\ (\sigma_1{\downarrow}1)\ (\sigma_1{\downarrow}2)\ \rho_1\ \iota\ \zeta_1 \\
[\![\texttt{tf}]\!] = \\
\quad \begin{cases} [\![\texttt{f}_1]\!] & \text{if } [\![\texttt{f}_1]\!] \in V; \\ [\![\texttt{(f}_1\texttt{)}]\!] & \text{otherwise.} \end{cases}
\end{array}
$$

User-defined function application is also very similar to the corresponding equation defined for $C'$. However, $C''$ pushes $[\![\texttt{e}]\!]$ onto the argument stack, so that it is eventually accessible to the lambda expression represented by $[\![\texttt{f}]\!]$. After compilation of the function is complete, the compiler analyzes the argument with the top element of the pattern stack returned by the analysis of the function.

($C''$ 11)

$C\ [\![\texttt{id}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota\ \zeta =$

$$\begin{cases} \textbf{Variable} & \text{if } \textit{b-type} = \textbf{lambda}; \\ \textbf{Reached-Limit} & \text{if } (\textit{pa}\ \alpha{\cdot}\pi) = \textit{unbound} \textbf{ and } \textit{v-count} \geq \iota; \\ \textbf{Compile-Binding} & \text{if } (\textit{pa}\ \alpha{\cdot}\pi) = \textit{unbound} \textbf{ and } \textit{v-count} < \iota; \\ \textbf{Mark-With-Pattern} & \text{otherwise;} \end{cases}$$

**where** $\langle [\![\texttt{binding}]\!], \textit{pa}, \textit{v-count}, \textit{b-type}, \textit{var-type}\rangle = \rho\ [\![\texttt{id}]\!]$;

**Variable is**

$\textit{var-type} = \textbf{var} \longrightarrow$

$[\![\texttt{id}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho_1\ \iota\ \zeta$

**where**

$$\rho_1 = \lambda i. \begin{cases} \langle [\![\texttt{binding}]\!], (\alpha{\cdot}\pi \sqcup \textit{pa}) \sqcap \pi_0, \textit{v-count}, \textit{b-type}, \textit{var-type}\rangle, \\ \quad \text{if } i = [\![\texttt{id}]\!]; \\ \rho\ i, \text{otherwise;} \end{cases}$$

,

$C\ [\![\textit{Fresh}\ ((\rho\ [\![\texttt{id}]\!])\downarrow 1)]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho\ \iota\ \zeta$

**Reached-Limit is** $[\![\texttt{binding}]\!]\ \alpha{\cdot}\pi\ (\textit{var-type} = \textbf{var} \longrightarrow \sigma, \bot)\ \rho\ \iota\ \zeta$;

**Compile-Binding is**

$[\![\texttt{fix:[id}|\alpha{\cdot}\pi\ \ \texttt{e}_1\texttt{]}]\!]\ \alpha{\cdot}\pi\ (\textit{var-type} = \textbf{var} \longrightarrow \sigma, \sigma_1)\ \rho_3\ \iota\ \zeta$

**where**

$[\![\texttt{fix:[id e]}]\!] = [\![\texttt{binding}]\!]$

$[\![\texttt{e}_1]\!]\ \alpha_1{\cdot}\pi_1\ \sigma_1\ \rho_1\ \iota_1\ \zeta_1 = C\ [\![\texttt{e}]\!]\ \alpha{\cdot}\pi\ \sigma\ \rho_2\ \iota\ \zeta$;

$$\rho_2 = \lambda i. \begin{cases} \langle [\![\texttt{fix:[id e]}]\!], \textit{pa}_2, \textit{v-count} + 1, \textbf{fix}, (\textit{Type}\ [\![\texttt{id}]\!])\rangle, \\ \quad \text{if } i = [\![\texttt{id}]\!]; \\ \rho\ i, \text{otherwise;} \end{cases}$$

$$\textit{pa}_2 = \lambda \textit{pat}. \begin{cases} \sigma_1 & \text{if } \textit{pat} = \alpha{\cdot}\pi; \\ (\textit{pa}\ \textit{pat}) & \text{otherwise;} \end{cases}$$

$$\rho_3 = \lambda i. \begin{cases} \rho\ i, & \text{if } i = [\![\texttt{id}]\!]; \\ \rho_1\ i, & \text{otherwise.} \end{cases}$$

**Mark-With-Pattern is**

$[\![\texttt{id}|\alpha{\cdot}\pi]\!]\ \alpha{\cdot}\pi\ (\textit{var-type} = \textbf{var} \longrightarrow \sigma, (\textit{pa}\ \alpha{\cdot}\pi))\ \rho\ \iota\ \zeta$.

Originally, it was assumed that lambda bound values could not be functions. This is no longer the case, which means that two possible ways in which to

compile variables are needed. If the variable's binding type is **lambda** and it is bound to a function value, or a list containing a function, then the compiler compiles a version of the expression to which it is bound, otherwise it behaves as in ($C'$ 11). *Fresh* renames instances of the formal in the lambda expression forming its argument; it's essentially alpha conversion.

The other three cases must handle two possibilities concerning the pattern stack returned by the compilation. Either the returned expression (identifier or binding) does represent a function, in which case a synthesized pattern is returned so that the function argument can be compiled, or the identifier does not represent a function, in which case there is no new synthesized pattern to be returned.

### Section 5.4: Extended examples

The following examples become increasingly complex in order. The notation $n \longrightarrow exp$ points to steps in the compilation process; not all steps are included). A similar notation $r \longleftarrow exp'$ indicates the result produced by $r \longrightarrow exp$. (In these examples, constants are not automatically marked wherever they appear; instead they are treated as any other expression, so that it is easy to see the effect of the propagation of synthesized patterns.)

**Example:** $[\![$(λa. head:a):<1 . 2>$]\!]$

$1 \longrightarrow C''\ [\![$(λa. head:a):<1 . 2>$]\!]\ [\![]\!]\ \$\bot\ []\ \rho^{init}\ \iota\ [] = \cdots$

$2 \longrightarrow C''\ [\![$head : a$]\!]\ \$\bot\ []\ \rho\ \iota\ [\ [\![< 1 . 2 >]\!]\ ] = \cdots$

$3 \longrightarrow C''\ [\![$a$]\!]\ \$\langle \$\bot\ ,\ \bot \rangle\ []\ \rho\ \iota\ [\ [\![< 1 . 2 >]\!]\ ] = \cdots$

$3 \longleftarrow [\![$a$]\!]\ \$\langle \$\bot, \bot \rangle\ []$
$\rho[$a$/\langle [\![< 1 . 2 >]\!], \$\langle \$\bot, \bot \rangle, 0,$ **lambda, var**$\rangle]\ \iota\ [\ [\![< 1 . 2 >]\!]\ ] \cdots$

$2 \longleftarrow [\![$head : a$]\!]\ \$\bot\ []$
$\rho[$a$/\langle [\![< 1 . 2 >]\!], \$\langle \$\bot, \bot \rangle, 0,$ **lambda, var**$\rangle]\ \iota\ [\ [\![< 1 . 2 >]\!]\ ] \cdots$

$1 \longleftarrow [\![$(λa. head:a):<$1 . 2>$]\!]\ \$\bot\ []\ \rho^{init}\ \iota\ []$

**Example:** $[\![$(head:<$\lambda$c. inc:head:c . $\lambda$d. 1>):<g . h>$]\!]$

$1 \longrightarrow C''\ [\![$(head:<$\lambda$c. inc:head:c . $\lambda$d. 1>) :<g . h>$]\!]\ \$\bot\ \sigma\ \rho\ \iota\ \zeta = \ldots$

$2 \longrightarrow C''\ [\![$head:<$\lambda$c. inc:head:c . $\lambda$d. 1>$]\!]$
$\$\bot\ \sigma\ \rho\ \iota\ [\,[\![$<g . h>$]\!]\ \zeta] = \ldots$

$3 \longrightarrow C''\ [\![$<$\lambda$c. inc:head:c . $\lambda$d. 1>$]\!]$
$\$\langle\$\bot\ ,\ \bot\rangle\ \sigma\ \rho\ \iota\ [\,[\![$<g . h>$]\!]\ \zeta] = \ldots$

$3 \longleftarrow [\![$<$\$\lambda$c. inc:head:c . $\lambda$d. 1>$]\!]$
$\$\langle\$\bot, \bot\rangle\ \langle[\$\langle\$\bot, \bot\rangle\ \sigma], [\bot\ \sigma]\rangle\ \rho\ \iota\ [\,[\![$<g . h>$]\!]\ \zeta] = \ldots$

$2 \longleftarrow [\![$head:<$\$\lambda$c. inc:head:c . $\lambda$d. 1>$]\!]$
$\$\bot\ [\$\langle\$\bot, \bot\rangle\ \sigma]\ \rho\ \iota\ [\,[\![$<g . h>$]\!]\ \zeta] = \ldots$

$1 \longleftarrow [\![$(head:<$\$\lambda$c. inc:head:c . $\lambda$d. 1>) :<$\$$g . h>$]\!]\ \$\bot\ \sigma\ \rho\ \iota\ \zeta$

The following example shows why it is necessary to take the *meet* of the pattern trees produced by the compilation of each branch. Note that a function is returned by each branch, but one of these functions does not require its argument.

**Example:**
$[\![$(if:<p:1 $\lambda$a. <head:a . 1> $\lambda$b. <2 . 2>>):<c . <>>$]\!]$

$1 \longrightarrow C''\ [\![$(if:<p:1 $\lambda$a. <head:a . 1> $\lambda$b. <2 . 2>>):<c . <>>$]\!]$
$\langle\$\bot, \$\bot\rangle\ \sigma\ \rho\ \iota\ \zeta = \ldots$

$2 \longrightarrow C''\ [\![$if:<p:1 $\lambda$a. <head:a . 1> $\lambda$b. <2 . 2>>$]\!]$
$\langle\$\bot, \$\bot\rangle\ \sigma\ \rho\ \iota\ [\,[\![$<c . <>>$]\!]\ \zeta] = \ldots$

$2 \longleftarrow [\![$if:<$\$$p:1 $\lambda$a. <$\$$head:a . $\$$1> $\lambda$b. <$\$$2 . $\$$2>>$]\!]$
$\langle\$\bot, \$\bot\rangle\ [\bot\ \sigma]\ \rho\ \iota\ \zeta \ldots$

$1 \longleftarrow [\![$(if:<$\$$p:1 $\lambda$a. <$\$$head:a . $\$$1> $\lambda$b. <$\$$2 . $\$$2>>):<c . <>>$]\!]$
$\langle\$\bot, \$\bot\rangle\ \sigma\ \rho\ \iota\ \zeta$

**Extended example:**

This example illustrates the mechanism used to ensure that arguments are passed to the correct bound variables. Note the way in which arguments containing higher order functions are compiled. (In this example, constants such as [[]] are not marked so that it is easy to see the propagation of patterns to the compilation of sub-expressions.)

When evaluated, the example produces the list [[<b . 3>]].

$[\![$((λa. λf. <head:a . (head:f):<2 . <>>>) :<b . <>>):
<λc. inc:head:c . <>>$]\!]$

1 $\longrightarrow C''\,[\![$((λa. λf. <head:a . (head:f): <2 . <>>>):<b . <>>):
<λc. inc:head:c . <>>$]\!]$
$\langle \$\bot, \$\bot\rangle\ \sigma\ \rho\ \iota\ \zeta = \ldots$

2 $\longrightarrow C''\,[\![$(λa. λf. <head:a . (head:f):<2 . <>>>):<b . <>>$]\!]$
$\langle \$\bot, \$\bot\rangle\ \sigma\ \rho\ \iota\ [\,[\![$<λc. inc:head:c . <>>$]\!]\ \zeta] = \cdots$

3 $\longrightarrow C''\,[\![$λa. λf. <head:a . (head:f):<2 . <>>>$]\!]$
$\langle \$\bot, \$\bot\rangle\ \sigma\ \rho\ \iota\ [\,[\![$<b . <>>$]\!]\ [\![$<λc. inc:head:c . <>>$]\!]\ \zeta] = \cdots$

4 $\longrightarrow C''\,[\![$λf. <head:a . (head:f):<2 . <>>>$]\!]$
$\langle \$\bot, \$\bot\rangle\ \sigma\ \rho[\mathtt{a}/\langle [\![$<b . <>>$]\!], \bot, 0, \mathbf{lambda}, \mathbf{var}\rangle]$
$\iota\ [\,[\![$<λc. inc:head:c . <>>$]\!]\ \zeta] = \cdots$

5 $\longrightarrow C''\,[\![$<head:a . (head:f):<2 . <>>>$]\!]\ \langle \$\bot, \$\bot\rangle\ \sigma$
$\rho[\mathtt{f}/\langle [\![$<λc. inc:head:c . <>>$]\!], \bot, 0, \mathbf{lambda}, \mathbf{function}\rangle]$
$[\mathtt{a}/\langle [\![$<b . <>>$]\!], \bot, 0, \mathbf{lambda}, \mathbf{var}\rangle]\ \iota\ \zeta = \cdots$

For the next few lines,
$\rho' = \rho[\mathtt{f}/\langle [\![$<λc. inc:head:c . <>>$]\!], \bot, 0, \mathbf{lambda}, \mathbf{function}\rangle]$
$[\mathtt{a}/\langle [\![$<b . <>>$]\!], \$\langle \$\bot, \bot\rangle, 0, \mathbf{lambda}, \mathbf{var}\rangle]$
$\rho'' = \rho[\mathtt{f}/\langle [\![$<λc. inc:head:c . <>>$]\!], \bot, 1, \mathbf{lambda}, \mathbf{function}\rangle]$
$[\mathtt{a}/\langle [\![$<b . <>>$]\!], \$\langle \$\bot, \bot\rangle, 0, \mathbf{lambda}, \mathbf{var}\rangle]$

6 $\longrightarrow C''\,[\![$(head:f):<2 . <>>$]\!]\ \$\bot\ \sigma\ \rho'\ \iota\ \zeta = \cdots$

7 $\longrightarrow C''\,[\![\mathtt{head:f}]\!]\ \$\bot\ \sigma\ \rho'\ \iota\ [\,[\![$<2 . <>>$]\!]\ \zeta] = \cdots$

8 $\longrightarrow C''\,[\![\mathtt{f}]\!]\ \$\langle \$\bot, \bot\rangle\ \sigma\ \rho'\ \iota\ [\,[\![$<2 . <>>$]\!]\ \zeta] = \cdots$

9 $\longrightarrow C''\,[\![$<λc. inc:head:c . <>>$]\!]\ \$\langle \$\bot, \bot\rangle\ \sigma\ \rho''\ \iota\ [\,[\![$<2 . <>>$]\!]\ \zeta] = \cdots$

```
9 ⟵ ⟦<$λc. inc:head:c . <>>⟧
      $⟨$⊥, ⊥⟩ ⟨[$⟨$⊥, ⊥⟩ σ], σ⟩ ρ'' ι [⟦<2 . <>>⟧ ζ]...

8 ⟵ ⟦<$λc. inc:head:c . <>>⟧
      $⟨$⊥, ⊥⟩ ⟨[$⟨$⊥, ⊥⟩ σ], σ⟩ ρ'' ι [⟦<2 . <>>⟧ ζ]...

7 ⟵ ⟦head:<$λc. inc:head:c . <>>⟧
      $⊥ [$⟨$⊥, ⊥⟩ σ] ρ'' ι [⟦<2 . <>>⟧ ζ]...

6 ⟵ ⟦(head:<$λc. inc:head:c . <>>):<$2 . <>>⟧ $⊥ σ ρ'' ι ζ...

5 ⟵ ⟦<$head:a . (head:<$λc. inc:head:c . <>>):<$2 . <>>>⟧
      ⟨$⊥, $⊥⟩ σ ρ'' ι ζ...

4 ⟵ ⟦λf. <$head:a . (head:<$λc. inc:head:c . <>>):<$2 . <>>>⟧ ι ζ...
    ⟨$⊥, $⊥⟩ [⊥ σ] ρ[a/⟨⟦<b . <>>⟧, ⊥, $⟨$⊥, ⊥⟩, 0, lambda, var⟩] ι ζ...

3 ⟵ ⟦λa. λf. <$head:a . (head:<$λc. inc:head:c . <>>):
    <$2 . <>>>⟧ ⟨$⊥, $⊥⟩ [$⟨$⊥, ⊥⟩ ⊥ σ] ρ ι ζ...

2 ⟵ ⟦(λa. λf. <$head:a . (head:<$λc. inc:head:c . <>>):
    <$2 . <>>>):<$b . <>>⟧
    ⟨$⊥, $⊥⟩ [⊥ σ] ρ ι ζ...

1 ⟵ ⟦((λa. λf. <$head:a . (head:<$λc. inc:head:c . <>>):
    <$2 . <>>>:<$b . <>>):<λc. inc:head:c . <>>⟧
    ⟨$⊥, $⊥⟩ σ ρ ι ζ
```

This final example steps through the compilation of a *fix* expression returned as a value. When evaluated, this program produces a stream of alternating successive increments of 3 and 80, or

⟦<3 80 4 81 5 82 6 83...>⟧.

let $[\![\texttt{exp}]\!]$ =
$[\![$((fix:[g $\lambda$n. $\lambda$p. <head:n . <head:p .
(g:<inc:head:n . <>>):<inc:head:p . <>>>>]):
<3 . <>>):<80 . <>>$]\!]$

$1 \longrightarrow C''\ [\![\texttt{exp}]\!]\ fix\,\lambda\pi.\langle \$\bot\ ,\ \langle \bot\ ,\ \pi\rangle\rangle\ \sigma\ \rho\ \iota\ \zeta = \ldots$

For the next few lines,
$rec\text{-}p = [\$\langle \$\bot, \bot\rangle\ \bot\ \sigma]$

$pa' = \lambda\ pat.\begin{cases} rec\text{-}p & \text{if } pat = fix\,\lambda\pi.\langle \$\bot\ ,\ \langle \bot\ ,\ \pi\rangle\rangle; \\ unbound & \text{otherwise;} \end{cases}$

$\rho' = \rho[\,[\![\texttt{g}]\!]\,/\langle [\![\lambda\texttt{n. }\lambda\texttt{p. <head:n . <head:p . (g:<inc:head:n . <>>):}$
$\texttt{<inc:head:p . <>>>>}]\!], pa', 0, \textbf{fix}, \textbf{function}\rangle]$

$\rho'' = \rho'[\,[\![\texttt{n}]\!]\,/\langle [\![\texttt{<3 . <>>}]\!], \bot, 0, \textbf{lambda}, \textbf{var}\rangle]$
$[\,[\![\texttt{p}]\!]\,/\langle [\![\texttt{<80 . <>>}]\!], \bot, 0, \textbf{lambda}, \textbf{var}\rangle]$

$\rho''' = \rho''[\,[\![\texttt{n}]\!]\,/\langle [\![\texttt{<3 . <>>}]\!], \$\langle \$\bot, \bot\rangle, 0, \textbf{lambda}, \textbf{var}\rangle]$
$[\,[\![\texttt{p}]\!]\,/\langle [\![\texttt{<80 . <>>}]\!], \bot, 0, \textbf{lambda}, \textbf{var}\rangle]$

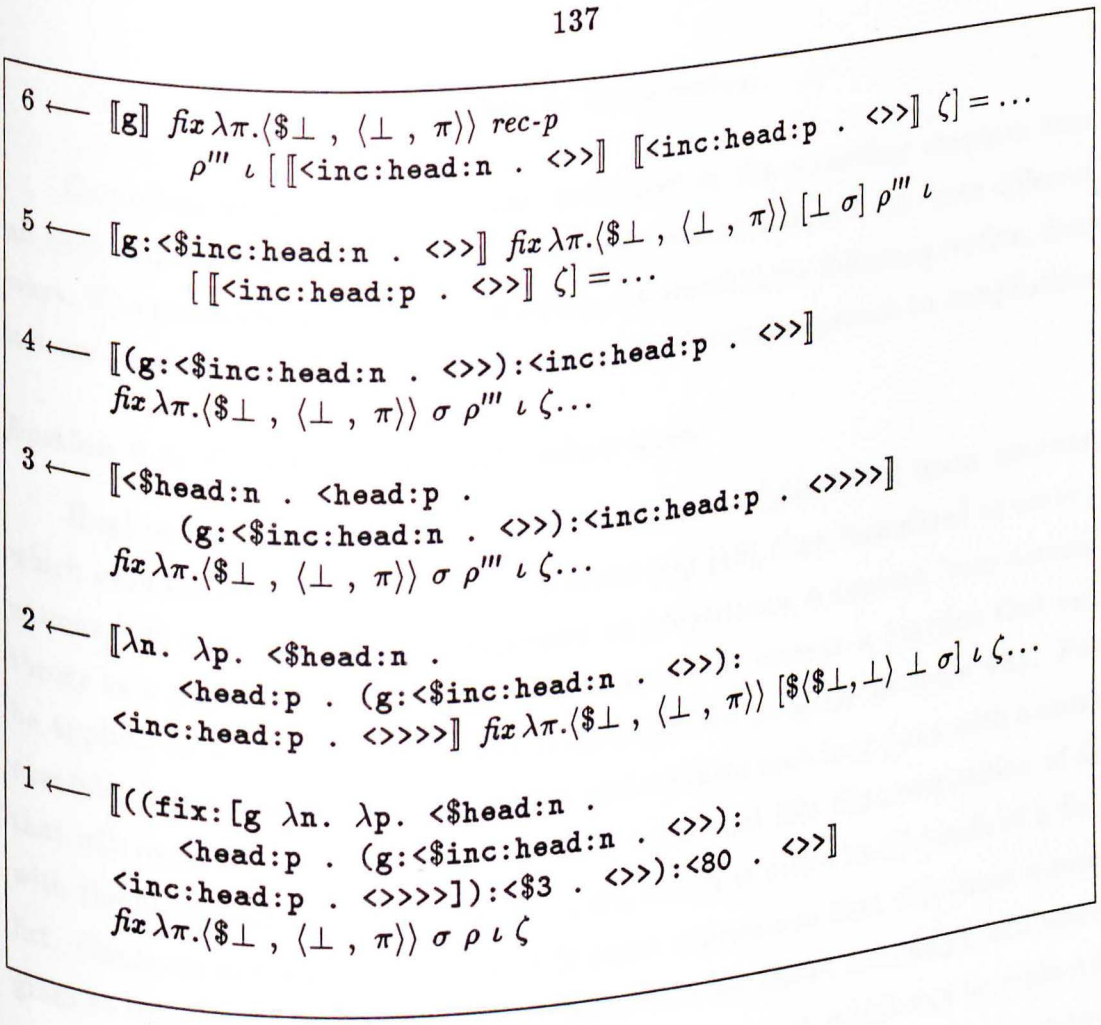
$2 \longrightarrow C''\ [\![\lambda\texttt{n. }\lambda\texttt{p. <head:n .}$
$\texttt{<head:p . (g:<inc:head:n . <>>):<inc:head:p . <>>>>}]\!]$
$fix\,\lambda\pi.\langle \$\bot\ ,\ \langle \bot\ ,\ \pi\rangle\rangle\ \sigma\ \rho'\ \iota\ [\,[\![\texttt{<3 . <>>}]\!]\ [\![\texttt{<80 . <>>}]\!]\ \zeta] = \ldots$

$3 \longrightarrow C''\ [\![\texttt{<head:n . <head:p .}$
$\texttt{(g:<inc:head:n . <>>):<inc:head:p . <>>>>}]\!]$
$fix\,\lambda\pi.\langle \$\bot\ ,\ \langle \bot\ ,\ \pi\rangle\rangle\ \sigma\ \rho''\ \iota\ \zeta = \ldots$

$4 \longrightarrow C''\ [\![\texttt{(g:<inc:head:n . <>>):<inc:head:p . <>>}]\!]$
$fix\,\lambda\pi.\langle \$\bot\ ,\ \langle \bot\ ,\ \pi\rangle\rangle\ \sigma\ \rho'''\ \iota\ \zeta = \ldots$

$5 \longrightarrow C''\ [\![\texttt{g:<inc:head:n . <>>}]\!]\ fix\,\lambda\pi.\langle \$\bot\ ,\ \langle \bot\ ,\ \pi\rangle\rangle\ \sigma\ \rho'''\ \iota$
$[\,[\![\texttt{<inc:head:p . <>>}]\!]\ \zeta] = \ldots$

$6 \longrightarrow C''\ [\![\texttt{g}]\!]\ fix\,\lambda\pi.\langle \$\bot\ ,\ \langle \bot\ ,\ \pi\rangle\rangle\ \sigma\ \rho'''\ \iota$
$[\,[\![\texttt{<inc:head:n . <>>}]\!]\ [\![\texttt{<inc:head:p . <>>}]\!]\ \zeta] = \ldots$


6 ⟵ ⟦g⟧ fix λπ.⟨$⊥ , ⟨⊥ , π⟩⟩ rec-p
ρ''' ι [ ⟦<inc:head:n . <>>⟧ ⟦<inc:head:p . <>>⟧ ζ] = ···
5 ⟵ ⟦g:<$inc:head:n . <>>⟧ fix λπ.⟨$⊥ , ⟨⊥ , π⟩⟩ [⊥ σ] ρ''' ι
[ ⟦<inc:head:p . <>>⟧ ζ] = ···
4 ⟵ ⟦(g:<$inc:head:n . <>>):<inc:head:p . <>>⟧
fix λπ.⟨$⊥ , ⟨⊥ , π⟩⟩ σ ρ''' ι ζ···
3 ⟵ ⟦<$head:n . <head:p .
(g:<$inc:head:n . <>>):<inc:head:p . <>>>>⟧
fix λπ.⟨$⊥ , ⟨⊥ , π⟩⟩ σ ρ''' ι ζ···
2 ⟵ ⟦λn. λp. <$head:n .
<head:p . (g:<$inc:head:n . <>>):
<inc:head:p . <>>>>⟧ fix λπ.⟨$⊥ , ⟨⊥ , π⟩⟩ [$⟨$⊥,⊥⟩ ⊥ σ] ι ζ···
1 ⟵ ⟦((fix:[g λn. λp. <$head:n .
<head:p . (g:<$inc:head:n . <>>):
<inc:head:p . <>>>>]):<$3 . <>>):<80 . <>>⟧
fix λπ.⟨$⊥ , ⟨⊥ , π⟩⟩ σ ρ ι ζ

# Chapter 6: Conclusion

Compilers very similar to those developed in the preceding chapters have all been implemented; in fact *C* has been implemented in at least three different ways. The research of Wadler and Hughes, discussed in the following section, does not use versions and is based upon a far more abstract approach to compilation.

## Section 6.1: Comparisons with other work

Hughes has developed a form of strictness analysis based upon *contexts*, which were initially described in an intuitive way [18], then formalized as continuations [16] and most recently appeared as projections, a concept from domain theory in a paper with Wadler [38]. A context is in essence a function that can be applied to a program, and that transforms the program in some way. For example, H is a context that takes a list and replaces each lazy **cons** with a **cons** that will evaluate its first argument. H is somewhat like the composition of *C* with the strictness pattern $fix\,\lambda\pi.\langle\$\bot\ ,\ \pi\rangle$, which is strict in all heads of a flat list. Contexts are also used to identify some expressions that will cause a program to abort, allowing a compiler to substitute an **abort** command, and some expressions that will not be required by a function at all, which can be replaced with a dummy expression. This is useful extra information that is not provided by any of the compilers presented here. Hughes [18], along with Lindstrom [25], also proposes an initial context very similar to $\pi_0$. This seems to be a natural conclusion arrived at by researchers interested in propagating information "backwards" [17].

Wadler and Hughes create a finite domain of contexts that is oriented differently from **P**; the *top* element specifies a context to be used when it is not known whether a given function is strict in its argument, and so the argument is left alone to be evaluated lazily. One of the advantages of this representation is that it is always possible to find a fixed point, and that a minimum of exploration of the lattice is needed to find it. They present a finite domain of contexts over flat

lists whose elements (omitting ABS and FAIL, which are "absent" and "abort" contexts) appear to correspond to sets of strictness patterns in the following way: (Note that $\{p \mid p = fix\,\lambda\pi.\langle\pi,\bot\rangle[\bot/\pi]\}$ is the set of all patterns produced by substituting $\bot$ for $\pi$ in unfoldings of $fix\,\lambda\pi.\langle\pi,\bot\rangle$)

$$STR \equiv \{p \mid p = \$fix\,\lambda\pi.\langle\bot,\pi\rangle[\bot/\pi]\} \cup \{\$fix\,\lambda\pi.\langle\bot,\pi\rangle\}$$
$$H' \equiv \{p \mid p = \$fix\,\lambda\pi.\langle\$\bot,\pi\rangle[\bot/\pi]\} \cup \{\$fix\,\lambda\pi.\langle\$\bot,\pi\rangle\}$$
$$T' \equiv \{p \mid p = \$fix\,\lambda\pi.\langle\bot,\$\pi\rangle[\bot/\pi]\}$$
$$H' \sqcap T' \equiv \{p \mid p = \$fix\,\lambda\pi.\langle\$\bot,\$\pi\rangle[\bot/\pi]\}$$
$$ID \equiv \{p \mid p = fix\,\lambda\pi.\langle\bot,\pi\rangle[\bot/\pi]\} \cup \{fix\,\lambda\pi.\langle\bot,\pi\rangle\}$$
$$H \equiv \{p \mid p = fix\,\lambda\pi.\langle\$\bot,\pi\rangle[\bot/\pi]\} \cup \{fix\,\lambda\pi.\langle\$\bot,\pi\rangle\}$$
$$T \equiv \{p \mid p = fix\,\lambda\pi.\langle\bot,\$\pi\rangle[\bot/\pi]\}$$
$$H \sqcap T \equiv \{p \mid p = fix\,\lambda\pi.\langle\$\bot,\$\pi\rangle[\bot/\pi]\}$$

This domain contains single points that represent infinite chains in **P**, so the notation used is certainly very powerful. However, it is not as expressive as **P** (as presented in Chapter 2) because there is no way to represent strictness in arbitrary sublists, or subtrees, or even regular patterns of strictness, such as strictness in alternate heads or tails. The fact that the construction of **P** requires strictness of successively longer lists to be represented as chains rather than as a single point becomes an advantage, because elements of these chains can be connected up in a very general way, allowing strictness in some sublists but not in others. Some important functions do require arbitrary pieces of their list arguments. (For example, a few years ago I implemented a small terminal server in which each piece of information was tagged with a terminal id. It was essential to check the tags, but not always necessary to look at the information that was tagged.) In addition, lists are used, even in a strongly typed language, as general purpose grouping structures. It is their generality that makes them essential to a wide and diverse variety of programs. The fact that they were used to group function arguments together as well as to create those argument values is crucial to the insight that allowed Friedman and Wise [11] to create a lazy

language essentially just by altering **cons**. For these reasons, it seems best to describe strictness in lists as generally as possible.

Fairbairn and Wray [9] discuss the use of versions in compiling higher order functions without list structures, also discussed in Wray's thesis [41]. They then go on to say that

> For this scheme to yield significant results it would be necessary to use list strictness analysis as well. This is because in the body of *map* there is the expression *cons* (*function* (*head list*)) (*map function* (*tail list*)) and (*function* (*head list*)) would still have to be constructed as an application tree if the standard lazy **cons** were used. With list strictness analysis [Wadler 85, for example] it would be possible to determine that (say) the whole output list of *map* would be required, so a strict version of *cons* could be used. However, to do this there would have to be even more versions of *map*, each with a different kind of *cons*, or else strictness information would have to be passed around at run-time to enable map to take appropriate action. We believe that the first alternative may lead to an unacceptable increase in code size, but the second scheme appears to be quite promising.[9, p. 100]

This work has demonstrated that there are interesting programs that would be greatly improved by versions, which do not increase the code size unacceptably in such cases, and that the user can severely restrict the number of versions created for each function if an increase in code size is a special concern.

Assume that *map* is defined to produce an infinite list, so that the body of *map* is just the **cons** expression discussed above. There are two possibilities. Either the call to *map* inherits a finite pattern, or it inherits a cyclic pattern. If the pattern is finite, then there is indeed a danger that a large number of versions of *map* may be produced; this would happen if the user gave the compiler a large resource and the finite pattern was a long one, allowing the compiler to unroll *map* several times. If the pattern is infinite, then the number of versions depends

upon the cycle length of the pattern and the user's tolerance; either the compiler would be allowed to create a series of versions that referred to each other in a cycle, or it would create a smaller number of versions but fail to close the cycle. However, if the cycle is successfully closed, then versions take up a constant amount of space and avoid an unbounded number of suspensions, a number that may be very large when infinite lists are prominent data structures in the definition of the function.

**Section 6.2: Contribution of research presented here**

The work presented here is based upon several straightforward ideas that interact in a fruitful way. For example, the domain of strictness patterns is expressive. Strictness in any list or sublist may be represented in the lattice **P** of strictness patterns. *C* is able to take advantage of this expressiveness without propagating patterns that would cause it to loop indefinitely. For example, the result of a function need not be a list that is consumed in a "homogeneous" way (all heads or all tails consumed) in order for the compiler to produce an appropriate version. One of the advantages of the pattern notation presented here is that *C* need not compile the leaves of the source code tree with patterns that exactly match their type; instead the patterns can be propagated to arbitrary points of the tree, where they are truncated if necessary. This allows the compiler to "lazily" match the depth of the strictness pattern inherited by the compilation of an arbitrary tree structure to the depth of that structure, rather than determine a set of available patterns before compilation.

The efficiency offered by versions in loops, especially loops that produce streams, make them worth exploring. The central loop in a program might be compiled into a cycle of twenty versions, permitting five suspensions to be avoided each time the loop were executed, causing an acceptable increase in code size simply because the speed of this loop is vital. However, when versions are *not* created, the *meet* of the patterns inherited by the set of applications of a given function is the only pattern that can safely be used to compile the function.

As has been shown in Chapter 2, this pattern can be very weak even though each pattern inherited would have produced an efficient version, which produces especially poor object code for functions that inherit cyclic patterns and produce infinite lists.

Some care must be taken when creating versions. For example, once the domain of source expressions includes functions that produce infinite lists, it is possible to generate an infinite number of versions for any of these functions because the result of their application can be consumed in an infinite number of different ways. A compiler that terminates must decide upon a finite number of versions that it will introduce into a given program. The interesting question is— how will it generate these versions? One approach might be to create a set using brute force, and then determine which, if any, will be useful. A better solution is found by $C$, which lazily creates versions on the fly when needed.

It may seem mysterious that $C$ limits the number of versions created for any one function when $C$ generates only versions actually required in compiling a program and propagates either finite or cyclic patterns. Finite patterns cause the generation of a finite number of versions, and since strictness patterns can be represented with a normal form, cyclic patterns have cycles which can eventually be recognized, at which point a reference to a previously constructed version can be introduced into the target code. The problem is that it is still possible for the compiler to execute a loop in which it propagates patterns that grow ever larger to recursive calls, so that each inherits a finite pattern different from its predecessor. It can also be useful to limit versions because the relationship between a function definition and the pattern inherited by its compilation is such that a sequence of several versions are created where each has little to offer.

To summarize, versions combined with the expressive power of **P** permit $C$ to receive and propagate patterns which avoid unnecessary approximation to a significant extent, and to produce target code which fully profits from this information. Applications which require efficient stream construction, such as a functional operating system or circuit simulation, are especially likely to benefit.

## Section 6.3: Areas for future investigation

The ideal strictness compiler would produce as many versions as needed, subject to a reasonable resource, but would then coalesce versions according to certain criteria. It can be the case that the same piece of code results from compilation of a function whose applications inherit a variety of strictness patterns. References to these versions can be compiled as references to only one distinct version. It may also be possible to develop techniques for selectively weakening versions where there is little point in introducing extra efficiency, such as sections of code that are not often executed. (Determining code that does little work is a notoriously difficult problem for programmers [22].)

The compilers presented here do not permit fine-tuning, in the sense that it isn't possible to use one resource in producing versions of $f$ and another to produce versions of $g$. This is an interesting area for future research.

Another area in which more work needs to be done involves finding pattern fixed points. Currently, the compiler may be forced to propagate an unnecessarily lazy pattern when its search for a pattern fixed point fails. The technique outlined in Chapter 2 (Section 2.5) works very well and fails gracefully (safely), but without further work its power can't really be compared to that of other methods. However, it does handle the examples presented in the preceding chapters, and seems to be potentially a powerful technique.

In many cases, finite lists can be detected and should be marked as strict. For example, function arguments are often collected by a finite list which can safely be marked. This particular improvement can be easily added to the compilers presented here, and there are probably many more such. Also, it is sufficient to ensure that only cyclic patterns are bounded by $\pi_0$—this is less restrictive than the constraint introduced by $C$ on synthesized patterns.

It would be worthwhile investigating what happens when the notion of buffering is introduced. At present, $\pi_0$ can be seen as a buffer of length one, but it is worth considering an altered printer pattern that is strict in $n$ tails, but repeats in an unmarked tail. For example, the pattern

$$fix\,\lambda\pi.\langle\$\pi, \$\langle\$\pi, \$\langle\$\pi, \pi\rangle\rangle\rangle$$

has a buffer of length three. This would allow more than just one element of a stream to be evaluated at a time, but would require that the user accept the loss of a buffer load of stream elements if any of those elements is $\perp_S$, a situation accepted by users of most conventional operating systems.

## Bibliography


1. S. Abramsky. Strictness analysis and polymorphic invariance. In Ganzinger and Jones (eds.), *Programs as Data Objects*, Springer, Berlin (October 1985), 1–23.

2. L. Augustsson. A compiler for lazy ML. *Conf. Rec. of the 1984 ACM Symposium on Lisp and Functional Programming*, (August 1984), 218–227.

3. A. Bloss and P. Hudak. Variations on strictness analysis. *ACM Conf. on Lisp and Functional Programming*, (August 1986), 132–142.

4. G. L. Burn, C. L. Hankin and S. Abramsky. Strictness analysis for higher order functions. In *Science of Computer Programming* 7, (1986), 249–278.

5. L. Cardelli, P. Wegner. On understanding types, data abstraction, and polymorphism. In *Computing Surveys* **7**, 4, (December 1985), 471–522.

6. R. Cartwright, J. Donahue. The semantics of lazy (and industrious) evaluation. *ACM Conf. on Lisp and Functional Programming*, (August 1982), 253–264.

7. C. Clack and S. L. Peyton-Jones. Generating parallelism from strictness analysis. In L. Augustsson, J. Hughes, T. Johnsson and K. Karlsson, (eds.), *Proceedings of Workshop on Implementation of Functional Languages* Report 17, Programming Methodology Group, University of Goteborg and Chalmers University of Technology, (February, 1985), 92–131.

8. C. Clack and S. L. Peyton Jones. Strictness analysis—a practical approach. In J. Jouannaud, (ed.), *Functional Programming Languages and Computer Architecture; Lecture Notes in Computer Science* **201**, Springer, Berlin (1985), 35–49.

9. J. Fairbairn and S. C. Wray. Code generation techniques for functional languages. *Proc. 1986 ACM Conference on Lisp and Functional Programming*, (August, 1986), 94–104.

10. J. Fairbairn. Ponder and its type system, Technical Report No. 31, University of Cambridge, Computer Laboratory (November 1982).

11. D. P. Friedman, and D. S. Wise. CONS should not evaluate its arguments. In S. Michaelson and R. Milner (eds.), *Automata, Languages and Programming*, Edinburgh University Press (1976), 257–284.

12. C. V. Hall and D. S. Wise. Compiling strictness into streams. In *Fourteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, (January, 1987), 132–143. Revised as Compiling strictness into streams. Tech. Report No. 209, Indiana University, (December, 1986).

13. C. V. Hall and J. T. O'Donnell. Debugging in a side effect free programming environment. *1985 ACM SIGPLAN Symposium on Programming Languages and Programming Environments*, *SIGPLAN Notices* **20**, 7, (June 1985), 60–68.

14. P. Henderson and J. H. Morris, Jr. A lazy evaluator. *Conf. Rec. 3rd ACM Symp. on Principles of Programming Languages* (January, 1976), 95–103.

15. P. Hudak and J. Young. Higher order strictness analysis for the untyped lambda calculus. *Conf. Rec. 13th ACM Symp. on Principles of Programming Languages* (January, 1986), 97–109.

16. R. J. M. Hughes. Analysing strictness by abstract interpretation of continuations. In S. Abramsky and C. Hankin, (eds.), *Abstract Interpretation of Declarative Languages*, Ellis Horwood, (in press).

17. R. J. M. Hughes. Backwards analysis of functional programs, March, 1987.

18. R. J. M. Hughes. Strictness detection in non-flat domains. In Ganzinger and Jones (eds.), *Programs as Data Objects*, Springer, Berlin (October 1985), 112-135.

19. S. D. Johnson, *Synthesis of Digital Designs from Recursion Equations*. M.I.T. Press, Cambridge, MA (1984).

20. S. D. Johnson, Daisy reference manual, Computer Science Dept. Technical Report Indiana University, Bloomington (in progress).

21. R. Kieburtz, M. Napierala. Abstract semantics. In S. Abramsky and C. Hankin, (eds.), *Abstract Interpretation of Declarative Languages*, Ellis Horwood, (in press).

22. D. E. Knuth. An empirical study of FORTRAN programs. In *Software—Practice and Experience*, 1, 105–133 (1971).

23. T. Kuo and P. Mishra. On strictness and its analysis. in *Fourteenth Annual ACM SIGACT—SIGPLAN Symposium on Principles of Programming Languages*, Munich, West Germany (January, 1987), 144–155.

24. P. J. Landin. A correspondence between ALGOL 60 and Church's lambda notation. *Comm. ACM* **8**, 2 (February, 1965), 89–101.

25. G. Lindstrom. Static evaluation of functional programs. *Proc. of the SIGPLAN '86 Symp. on Compiler Construction, SIGPLAN Notices* **21**, 7 (July 1986), 196–206.

26. D. Maurer. Strictness computation using lambda expressions. In Ganzinger and Jones (eds.), *Programs as Data Objects*, Springer, Berlin (October 1985), 136–155.

27. J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin. *Lisp 1.5 Programmer's Manual*, The MIT Press, Cambridge, Mass., 1973.

28. A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. *Proc. of Intl. Symp. on Programming, Lecture Notes in Computer Science* **83**, Berlin, Springer (1980), 269–281.

29. F. Nielson. Strictness analysis and denotational abstract interpretation. *Fourteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Munich, West Germany (January, 1987), 120–131.

30. J. T. O'Donnell and C. V. Hall. Debugging in applicative languages. Tech. Report No. 223, Indiana University Computer Science Department, June 1987.

31. J. T. O'Donnell. Hardware description with recursion equations. *Proc. of the IFIP 8th International Symposium on Computer Hardware Description Languages and their Applications*, North-Holland (April, 1987), 363–382.

32. J. T. O'Donnell. Dialogues: a basis for constructing programming environments. *Proc. of the ACM SIGPLAN 85 Symposium on Programming Languages and Programming Environments* (June, 1985), 19–27.

33. J. T. O'Donnell. Personal communication, December, 1986.

34. D. A. Schmidt. *Denotational Semantics, A Methodology for Language Development*, Allyn and Bacon, Newton, MA (1986) .

35. J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press (Cambridge, Mass., London) (1977).

36. D. A. Turner. Recursion equations as a programming language. In Darlington, Henderson and Turner (eds.), *Functional Programming and its Applications*, Cambridge University Press (1981) .

37. P. Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains), In S. Abramsky and C. Hankin, (eds.), *Abstract Interpretation of Declarative Languages*, Ellis Horwood, (in press).

38. P. Wadler and R. J. M. Hughes. Projections for strictness analysis. *Functional Programming Languages and Computer Architecture; Lecture Notes in Computer Science* **274**, Springer, Berlin (1987), 385–407.

. P. Wadler. Personal communication, February, 1987.

S. C. Wray. A new strictness detection algorithm. In L. Augustsson, J. Hughes, T. Johnsson and K. Karlsson, (eds.), *Proceedings of Workshop on Implementation on Functional Languages* (Aspenas, Goteborg, February

1985) Report 17, Programming Methodology Group, University of Goteborg and Chalmers University of Technology, 190–210.

41. S. C. Wray. *Implementation and programming techniques for functional languages*, PhD. Dissertation, University of Cambridge, June 1986.

## Vita

Cordelia Hall received a B. Mus. in Violin Performance from McGill University in 1978. She earned an M. Mus. from Indiana University in 1980 after study with Franco Gulli. From 1980 to 1981, she created a data base for research on patient profiles as a Graduate Assistant for the Indiana University Student Health Service. From 1981 to 1982 she was an Associate Instructor at the Indiana University Computer Science Department. Ms. Hall has been a Research Assistant at Indiana University from 1982 to 1986, and will be an S. E. R. C. Visiting Research Fellow at the University of Glasgow during 1987–1988. She has accepted a position as assistant professor at the University of Michigan.

Ms. Hall is a member of the Association for Computing Machinery.