On Linear Types and Imperative Update

by
Stephen Cooper

Abstract of Dissertation
August, 1997

The rift between imperative and functional programming is one of the oldest in computing. Imperative languages provide for greater speed and efficiency, especially through the use of destructive update of variables. Functional computation is often regarded as being much cleaner mathematically, and is supportive of simple, rigorous reasoning principles. However, the absence of destructive update (along with problems with garbage collection, etc.) in functional languages results in a loss of speed and efficiency. Additionally, functional languages are conceptually indirect and cumbersome when used to express naturally imperative computation. The prospect of unifying these diverse forms of programming, to combine the best features from each, is a continuing concern in programming languages. This thesis investigates the extent to, and the sense in which, programming languages based on linear logic are simultaneously imperative and purely functional, and to what extent they embody the strengths of each programming paradigm.

On Linear Types and Imperative Update

by

Stephen Cooper

BA

MS

Dissertation

Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer and Information Science
in the Graduate School of Syracuse University

August, 1997

Approved ______________________

Date ______________________

Acknowledgements

I would like to thank my advisor, Peter O'Hearn. No matter what was going on, he always had time for me, whether discussing some research issue, or just helping to understand some concept. Also, thank you Peter for letting me stay with you the numerous times I was in London.

Thanks to Jim Royer and Sue Older for their early suggestions and comments on early drafts of this thesis. Also, thanks to Wanda Dann for her comments, and for being a good friend throughout this whole process. Thanks to Dennis Cleary for reviewing this thesis for grammatical, typographical, and any other English errors.

Finally, I would like to thank my wife Sandi for her endless patience and tireless support, and my parents Jack and Billie for instilling in me the value of a good education.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this thesis, the imperative aspects of variations on the linear λ-calculus are studied. Linear λ-calculi arise from Girard's linear logic [Gir87], via the propositions-as-types correspondence (see e.g. [GLT89]). In this view, inference rules from logic are used to derive typing assertions about programs. The novelty of linear typing is that it arises from a refinement of traditional (intuitionistic) logic that, via the correspondence, leads to a refined form of functional programming. This form of functional programming is often described as being "resource conscious".

The introductory chapter covers three issues:

1. In Sections 1.1 and 1.2, we describe the problem tackled in this work.

2. In Section 1.3, we discuss related research.

3. In Section 1.4, we summarize our results.

Knowledge of the following topics is presumed: functional programming, the propositions-as-types correspondence [GLT89], elementary domain theory [Plo78], and operational semantics [Gun92].

1.1 Setting

In traditional logic, the structural rules of Weakening and Contraction are formulated as:

$$\frac{\Gamma \vdash A}{\Gamma, B \vdash A}\ \textsf{Weakening} \qquad \frac{\Gamma, B, B \vdash A}{\Gamma, B \vdash A}\ \textsf{Contraction}$$

Here, Weakening may be thought of as a means to discard resources without using them, and Contraction may be thought of as a means to copy them. Linear logic is a refinement of traditional logic that limits the use of the structural rules of Weakening and Contraction [Gir87]. When linear logic is used as a type system for a functional programming language, this control over structural rules translates into restrictions on the number of times a variable appears in the body of a function. The type $A \multimap B$ (pronounced A lollipop B, or A linear arrow B) of linear functions reflects these restrictions. Intuitively, linear functions use their arguments exactly once during their computation of a result.

There is an obvious analogy between linear functions and state transformations definable in imperative languages. Because there is no Contraction rule, a function of linear type cannot copy its argument. This is reminiscent of the idea that assignment overwrites computer memory, rather than making a copy. The old value in the computer's memory is simply destroyed. The connection between linear functions and Weakening is more difficult to establish. The idea is that a linear function cannot generally discard its argument (and in Weakening an assumption is not used/needed). As such, in linear functional programming there is, in general, no Weakening. The net result is the intuition about linear functions using their arguments exactly once.

As a concrete example, consider the following "snapback" operator:

$$\lambda s{:}\,\varphi.\lambda c{:}\,(\varphi \multimap (\varphi \otimes \mathbf{nat})).\mathbf{let}\ s' \otimes n\ \mathbf{be}\ cs\ \mathbf{in}\ s \otimes n.$$

Here, φ can be thought of as the type of the computer store, $\otimes$ as a linear-logic type constructor for a kind of pairing, and **nat** as the type of natural numbers. Snapback begins in a state s and uses c to change to another state, s', while computing a result, n. But then, after computing this result, the state is "snapped back" to its original value, s. In doing so, the state, s', is forgotten. Such functions are easily definable in conventional functional languages.

This reversal of state change runs counter to the imperative view, where a state transformation is understood as permanently changing the contents of the computer store. The problem with snapback is that it uses Contraction to make two copies of the state s, and Weakening to discard the altered state, s'. As Scott writes [Sco72],

> "...we will be tempted to copy ρ [the environment], but we will never generally feel free to ask for a copy of the *whole* computer store – there is just no room for that."

Linear functions may be used to prevent the copying of "the whole computer store".

The analogy between linear functions and imperative state transformations is one of the reasons for the great interest generated by linear logic in computer science. Functional computation is often regarded as being clean mathematically, and supportive of simple, rigorous reasoning principles. Imperative programming is frequently seen to be more realistic, especially as regards issues of space and time complexity, and natural expression of history-sensitive behavior. The analogy thus offers the hope of bridging the gap between these two forms of programming.

Of course, linear logic is not a panacea; a satisfactory marriage of functional and imperative programming is far from reality. A better understanding of its strengths and weaknesses is desirable, to help place linear typing in the design space of programming languages. This thesis proposes to further investigate this relationship between linear functions and destructive state changes.

Two forms of operational semantics are considered in this thesis for a functional language with types for cells (or arrays). One, a "functional operational semantics", is based on typical functional reductions and interprets array operations in terms of copying. The other, an "imperative operational semantics", uses an explicit state parameter and interprets array operations by changing the state using destructive overwrite.

We investigate situations where a coincidence result (see Section 1.4 for a more detailed description of coincidence) holds, to the effect that the functional and imperative operational semantics give equivalent behavior for well-typed programs.

Coincidence shows a sense in which a language can be read, or understood, in both functional and imperative terms: really, coincidence is a rigorous approximation to the intuitive idea that the language is "simultaneously imperative and purely functional". A series of languages based on linear typing is evaluated, obtaining both positive and negative results about coincidence, as well as extensionality properties of program equivalence and approximation (which express their functional character). Extensionality is discussed in more detail in Section 1.4.

1.2 Functions and Side Effects

The problem of integrating the functional and imperative forms of programming has remained a research issue for over thirty years. One symptom of the difficulty is the subtleties that arise when considering the issue of independence from evaluation order. We discuss this issue to exemplify some of the problems faced in this work, and to contrast the concepts underlying linear functional programming with other approaches to combining imperative and functional features.

A distinctive characteristic of pure functional languages is that the order of reductions does not affect the final result of a program. Because there is no state, an addition operation, for instance, can evaluate either its right or left argument first. Which of these it chooses to evaluate first does not affect the overall outcome. Adding side effects can break this property, as the following two examples illustrate.

Consider first the following code in Scheme:

```
(define x 0)
(define f (lambda (z) (+ z z)))

(f (begin (set! x (+ x 1))
           x)
```

)

If the value of the third expression is computed by evaluating the argument and then applying f to the result, 2 is returned as the final answer. If, instead, call-by-name evaluation is used, and the substitution for z occurs before evaluating the argument, 3 is returned as the final answer. Side effects typically destroy independence from evaluation order.

A similar example is the following code, written in a style similar to the languages we will consider:

$$\begin{array}{l}(\lambda x.\ \mathbf{let}\ a \otimes m\ \mathbf{be}\ \mathit{lookup}\ (\mathit{update}\ 7\ x)\ \mathbf{in}\\ \quad\ \mathbf{let}\ b \otimes n\ \mathbf{be}\ \mathit{lookup}\ (\mathit{update}\ 8\ x)\ \mathbf{in}\ m + n\\ )\ (\mathit{alloc}\ 0)\end{array}$$

Here, *alloc* is a function that allocates a single-element array and initializes it to the value specified (in this case, 0). The *update* function takes two arguments (an integer value and a single element array that is to be assigned that value) and changes the value of a single element array. The *lookup* function looks up the value of a single element array, returning the value of the element of the array in addition to the array itself. Depending on the order of evaluation, the result could be 14, 15, or 16. (Note that this program applies Contraction to identifier x, resulting in its two occurrences. It will be a valid pre-term, or untyped term, but will not be typable in the linear languages considered.)

This potential, and unruly, form of nondeterminism has been dealt with in a number of ways. Three of the most prominent are:

1. *Imposition of a specific evaluation order.* The language Scheme uses this technique. It specifies an "eager" evaluation strategy (also known as call by value) in which the argument is always evaluated first. In the first example, in Scheme, the argument would be evaluated, and x set to 1 before it is substituted for z. So, the net result of the expression would be 2.

2. *Separation of side effects from the procedure mechanism.* The most notable example of this technique is Idealized Algol [Rey81], where all side effects are

encapsulated in a type **comm** of commands. A similar use of types has recently become popular in the functional programming community, where monads are used to provide a form of parameterized command type [Wad92].

3. *Linear logic.*

The first alternative is the most widespread. It is present in such languages as Lisp, Scheme and ML. While it has substantial merits, we do not consider it to be a true integration of pure functional and imperative programming. Determinacy is achieved at the expense of the language's functional character. In particular, function types in these languages contain much more than mathematical functions from values to values, since they must allow for side effects, exceptions, and so on.

The second alternative is more functional in nature (see [WF93, O'H95]) but involves the separation of functions and state rather than their integration. For example, commands in Idealized Algol (or terms of monad type in other languages) are not essentially different from imperative mechanisms in standard imperative languages. They require typical imperative or state-based reasoning to be understood, although functional properties (such as β and η laws) are retained in the language. It can be argued that protecting functions from side effects thus retains some attractive aspects of functional reasoning, but it does not make the effects themselves substantially easier to understand. Finally, the standard context lemma (also called operational extensionality), which states that "function types consist of functions", fails for these languages [O'H95]. Pitts [Pit96] has shown an alternative, reasonably well-behaved, form of context lemma for Idealized Algol. However, its use of "possible worlds", corresponding to local storage variable declarations, highlights the difference with functional languages.

Linear logic, the third alternative, promises more of an integration of functional and imperative computation. Although there has been good progress, the consequences of linear logic for imperative and functional programming have not been worked out as thoroughly as the other two approaches. This line of work is considered in more detail in Section 1.3.

This account of languages that combine functional and imperative features is admittedly incomplete and biased. Justice has not been done to the ML/Scheme tradition. Neither has it been shown that having a purely functional language is ultimately desirable. (It would be difficult to give a definitive arguments for this.) Rather, features that distinguish the linear logic approach have been set forth. Independence from evaluation order is one way to quantify the difference. Equally important are denotational models, and extensionality properties of program equivalence. All three of these differences will be matters for concern throughout the course of the thesis.

1.3 Related Work: Linear Functional Programming

As previously mentioned, linear typing provides a refined function type, $\theta \multimap \theta'$. From the discussion so far, this kind of function type may appear excessively weak. It is impossible to even write the function $\lambda x \,.\, x * x$, which calculates the square of its argument, because the argument x would have to be used twice. In linear logic, there is a new type constructor !, pronounced "bang". Contraction and Weakening are allowed, but only for types of the form $!\theta$. This gives rise to a distinction between non-linear types, of the form $!\theta$, and linear types such as $\theta \multimap \theta'$. Values of non-linear type can be copied, discarded, and manipulated at will—as they are in conventional functional languages. By combining ! and $\multimap$, conventional function types can themselves be recovered, as $!\theta \multimap \theta'$. These ideas are well presented in Girard's original paper [Gir87] and in Abramsky's excellent article [Abr93].

Almost immediately after linear logic was discovered by Girard, there were a number of attempts to incorporate its ideas into a programming setting. Early approaches were based on abstract machines, in the style of the SECD machine or of the Categorical Abstract Machine.

Lafont, in [Laf88], developed a linear functional language, accompanied by a categorical semantics and an abstract machine, the Linear Abstract Machine. The Linear

Abstract Machine has a remarkable property: Every value has exactly one pointer to it. As a result of this, Lafont states that no garbage collector is required for the language. This result indicates that linear types give fine control over memory utilization.

This absence of a garbage collector is remarkable. But, one naturally wonders at what cost it is achieved. In particular, the model which gives rise to it relies on (what appear to be) computationally expensive assumptions. Lafont interprets Contraction in his abstract machine by copying. As a result, all values, and not just linear ones, have one pointer to them. Taken at face value, a program that uses a variable ten times would have to copy its value ten different times. This would be prohibitively expensive. More delicate notions of "copying" might be considered, such as that of Mackie [Mac94], but the benefits of such mechanisms remain to be demonstrated. Indeed, the conclusions of Mackie appear to be negative in this regard (meaning that even this delicate form of "copying" did not sufficiently limit the amount of copying required).

Furthermore, if one were to use the same implementation mechanism (copying often enough) in the fully typed λ-calculus, this "one-pointer property" would still be achieved, and the same argument in regards to garbage collection would still apply. As a result, the specific gain obtained from linear types (over intuitionistic types) is not evident. It is not that there is no gain — just that it is not self-evident from the specific results given with the Linear Abstract Machine.

That said, the influence of Lafont's paper should be emphasized. Some of the intuitions expressed about linearity, and its categorical formulation, have contributed greatly to subsequent developments. In particular, later researchers were keen to take up the questions of garbage collection and control over memory utilization.

A number of researchers depart from Lafont's point of view by interpreting Contraction as sharing rather than as copying. Contraction can be viewed as a rule for allowing multiple occurrences of an identifier, as follows:

$$\frac{\Gamma, y\colon \theta', z\colon \theta' \vdash M\colon \theta}{\Gamma, x\colon \theta' \vdash M[x/y, x/z]\colon \theta}\ \textsf{Contraction}$$

(It is sometimes given a special binding syntax instead of using substitution). A sharing interpretation would make different occurrences of an identifier denote the same value in an implementation, whereas copying would allow them to denote different copies of a value that might be subsequently altered or optimized differently.

Wadler, in particular, takes an approach based on a graph reduction model, where Contraction results in two pointers that point to the same value [Wad90, Wad91]. This is thought to be a reasonable choice (by much of the programming language community), based on experience with functional language implementation. Moreover, by using Girard's translation $\theta \rightarrow \theta' \equiv !\theta \multimap \theta'$ (where $\theta \rightarrow \theta'$ is the function type of intuitionistic functions, which need not be linear), this approach leads to an implementation closer to the graph-reduction implementations of lazy functional languages.

It would have been idyllic if this sharing approach had resulted in terms of linear type having the beneficial properties held by terms in Lafont's language, even if the properties would not have held for other (non-linear) terms. More precisely, one could hope for a situation where terms of linear type, such as $\theta \multimap \theta'$, would have exactly one pointer to them (the "one-pointer property"), but where terms of type $!\theta$ might have multiple pointers to them. If successful, such a scheme would reduce the need for garbage collection and allow destructive update, as envisioned by Lafont. Intuitively, if a linear value has one pointer to it, on the occasion of its single use its storage can immediately be reclaimed, or overwritten. Unfortunately, under his interpretation, Wadler showed this hope to be false. In particular, because of the Dereliction operation of type $!\theta \multimap \theta$ (Dereliction and the problems associated with it are discussed in much greater detail in Chapters 5 and 6), it is possible for a value of !-type to be converted to a non-!-type. This leads to a situation where there are linear values with multiple pointers to them (again, unless enough copying is done).

It may be argued that such a negative conclusion is due more to the model used than linear types themselves. However, since then, no one has given an operational model for linear logic with a "one-pointer property" for linear types without relying on excessive copying as described above. Wadler based his arguments on an informal model, but they do appear to be reasonably robust. In particular, Chirimar, Gunter, and Riecke [CGR96] defined a model based on reference counting and independently came to a similar conclusion. In their model, Contraction results in a reference count being incremented, and values of non-!-type are not guaranteed to have a reference count of one. One conclusion that they do make clear, which is also used by Wadler, is that the number of pointers to a value of non-!-type is never increased. This point will be discussed again in Section 1.4.

This section has not been exact as regards to credit for certain observations. Actually, the chain of events and ideas is not particularly clear to us. In particular, workers such as Abramsky [Abr93] , Mackie [Mac94] , Holmström [Hol88] , Lincoln and Mitchell [LM92] have been concerned with linear functional programming, and have made related observations and remarks, especially on issues of garbage collection, reference counts, and sharing. Not discussing their work in more detail is not meant to slight them. Simply, the early work of Lafont, and the reaction/analysis of Wadler and Chirimar-Gunter-Riecke, represent clear positions on what linearity can mean operationally. It is these positions with and from which this work is compared and contrasted.

One of the reactions to the above-mentioned subtleties in linear and non-linear types was to work with different languages, which were inspired by, but not exactly based on, linear logic (e.g. [GH90] and [BS93]). This thesis will not be concerned with any of these other languages, except for two considered by Wadler (which remain quite close to the original spirit of linear functional programming). The next section discusses these in the context of this thesis.

1.4 An Overview of This Thesis

Most previous operational models of linear λ-calculus are not explicitly oriented toward showing the imperative nature of a linear language. Rather, the focus of these operational models has been on issues such as garbage collection or optimization. The resulting models (e.g., based on graph reduction or reference counts) are well suited to handling these issues. There is a clear connection between these ideas and assignment. Chirimar, Gunter, and Riecke [CGR96] state this connection as follows:

> "...we know that the reference count of ℓ_1 does not exceed one.... This implies that it is safe to update y [which denotes ℓ_1] in place."

And Wadler [Wad90] writes:

> "Values belonging to a linear type must be used exactly once: like the world [program state], they can be neither duplicated nor discarded."

While the points of view expressed in previous work can be appreciated as coincident with this thesis, it appears that these authors approach linear functional programming from a functional-programming perspective. That is, linearity gives constraints on a functional language, which, in turn, allows certain optimizations to be performed. A more imperative perspective is taken in this dissertation. Our view is that linear logic holds promise as an alternative and novel way to express imperative computations. As such, its imperative flavor should be accounted for more directly than by using it to help with garbage collection or for optimizations. That is to say, its simultaneously functional and imperative character should be explained.

This simultaneously functional and imperative character is the subject matter of Chapters 2, 3, and 4. In Chapter 2, a system of Rudimentary Linear Types is introduced, based on the $\otimes, \multimap, \times$ fragment of intuitionistic linear logic, a fragment without "!". (Rudimentary Types are sometimes called IMALL, for intuitionistic, multiplicative, additive linear logic. However, Rudimentary Types do not include co-products.) Constants are given for allocating, deallocating, updating, and accessing storage cells. Additionally, a primitive looping construct, **repeat**, is provided. Chapter 2 presents the syntax for this basic language .

Chapter 3 is devoted to exploring the functional character of the language. A "functional operational semantics" is defined, based on typical reduction semantics for functional languages. This formulation has three important characteristics.

1. Reductions are allowed to occur in any order.

2. The operational semantics is shown to be adequate with respect to a natural denotational model, which is formulated using values and functions, with *no* states.

3. An operational extensionality theorem is proved.

Allowing reductions to occur in any order and being adequate with respect to an appropriate (stateless) denotational model are properties one should expect from a functional language. Furthermore, operational extensionality (analogous to the context lemma for PCF) shows that functions consist of just... mathematical functions, understood as functions in classical set theory. There are a range of other properties that one could consider (η laws, etc.). Again, the goal is not to attempt a complete or definitive list but a useful one to guide investigation of the various linear languages.

Chapter 4 is devoted to exploring the imperative character of the language. An "imperative operational semantics", which uses functional reductions together with an explicit store parameter, is defined. In the imperative semantics, operations for updating a cell are modeled by changing the store. (In the functional semantics, update is modeled simply by producing a new value.) The main result connecting the two interpretations is proved:

> **Coincidence**. For any typable program, the functional and imperative operational semantics give equivalent behavior.

Therefore, from our perspective, the job of linear typing is to guarantee that the functional and imperative semantics agree on typable terms. The coincidence property is a reasonable approximation to the informal notion that a language is simultaneously imperative and purely functional. However, it must be stressed that the coincidence property is not a general formulation of the intuitive notion. The formal notion depends especially on the specific operational semantics given. It is the precise properties that make sense in specific context that we consider relevant, rather than general definitions of terms like "purely functional" (all attempted general definitions we have seen are seriously defective).

This coincidence result is perhaps not unexpected. The point is that the operational semantics gives a crisp formal expression to the imperative character of the language based on Rudimentary Linear Types. Of course, rudimentary typing is rather limited, though perhaps not as limited as commonly thought. It includes a limited form of recursion, available via the **repeat** combinator, allowing the encoding of a variety of imperative algorithms. When combined with linear functions, the resulting language might be thought of as a nontrivial subset of Fortran77. Interestingly, the **repeat** combinator breaks the intuition about functions of linear type using their arguments exactly once. It does not, however, break coincidence. Coincidence is a formal property that concerns the imperative nature of the language. Although this intuition about linear functions is at times useful, it is not necessary to obtain an imperative reading of linear types.

Next, extensions of the language are considered, using the development for Rudimentary Types as a model. In Chapter 5, adding "!" is considered, together with standard reductions which come from categorical semantics. The result is that coincidence fails. There are typable programs with different behaviors under the functional and imperative semantics. The problem lies with unrestricted use of the linear logic rules of Dereliction and Promotion (like Dereliction, Promotion will be discussed in much more detail in Chapters 5 and 6), and indicates that the connection between linear functions and state transformations is more subtle that generally believed, and in need of careful formalization. This observation agrees with conclusions drawn by Wadler in [Wad90].

A fix proposed by Wadler was to give the constant for allocating cells an unusual type[1]

$$block\colon\ !(\mathbf{cell} \multimap \mathbf{cell} \otimes \mathbf{nat}) \multimap \mathbf{nat}.$$

The idea behind the *block* construct is that $block(\lambda c.C)$ creates a new cell and binds it to c. Recall that Chirimar, Gunter, and Riecke had shown that the reference count of a linear value never increases. The relevance of this observation here is, as Wadler

[1]This is not the exact type given by Wadler, but it suffices for the purpose of our discussion.

writes [Wad91],

> "...we cannot guarantee *a priori* that a variable of linear type has exactly one pointer to it. But if we know this by other means, then linearity guarantees that the pointer will not be duplicated or discarded."

The *block* construct is explicitly designed to be one of these "other means". Wadler remarked, without proof, that the the *block* construct allows destructive update to be done safely.

This chapter verifies that a language with *block* satisfies coincidence, as would be expected, using our operational models. Furthermore, the use of full "!" allows some of the difficulties observed later for steadfast types (the second extension to Rudimentary Types considered) to be overcome. However, there is something unsettling about the type of *block*. Wadler simply called it "too unwieldy for convenient use" [Wad93], and it does seem unusual from a functional programming point of view. The following observation perhaps underlies this discomfort: operational extensionality fails in this language. As a result, it is questionable whether the language should be regarded as truly functional, since the failure of operational extensionality indicates that there is more to function types than (classical) functions.

In Chapter 6, the system of Steadfast Types, (a variation on linear logic Wadler designed in response to the difficulties with "!") is discussed. In Steadfast Types, the rules of Dereliction and Promotion are restricted. He claims that, in the resulting language, values of linear type have exactly one pointer to them, but again uses an informal model and analysis to justify the claim. Somewhat surprisingly, when we formulated a rigorous model we determined that steadfast types appear to suffer from essentially the same problem that Wadler observed in [Wad91], but using different examples to illustrate the problem.

Next, a modified version of Steadfast Types is designed to solve the problems found. The modification is surprisingly delicate, requiring that the type system and reduction rules be designed simultaneously with a proof of coincidence. The result is a language with full intuitionistic function types, together with linear types, in which coincidence and extensionality do hold. However, there are serious limitations that remain in the language, especially in the treatment of recursion. These limitations, it turns out, can also be seen in other type systems loosely inspired by linear logic (such as in [GH90] and [BS93]), though they do not seem to have been previously mentioned in the literature.

In Chapter 7, Rudimentary Types are revisited. A cost model for the language is considered. If a language can faithfully express imperative computations, it should have a clear cost model that assigns costs comparable to those in a conventional imperative language. The sense of "should" indicates an ideal, which here is a difficult general aim, but one that is relevant for properly measuring any attempt to integrate functional and imperative programming. A simple model is defined, basing the decisions of the costs of various operations on call-by-value reductions. Crucially, the operations of array update and lookup are assigned unit cost. This assignment is justified by our coincidence theorem.

To illustrate our model, Tarjan's algorithm for depth-first search through a graph is considered. Implementing depth-first search in the language of Rudimentary Types may be accomplished in $O(v+e)$, the same running time as the traditional, imperative version. The algorithm uses destructive update in an essential way. Graph search has been a leading example of a problem that is difficult to treat in a purely functional language, and no previous purely functional algorithm has been given that is $O(v+e)$. The point of the example is not to argue for the language, but to illustrate one attribute certainly expected of a language that is simultaneously imperative and purely functional.

Finally, Chapter 8 consists of a brief summary of our major results. Additionally, so future possible directions are considered.

Chapter 2

Rudimentary Types

The first step of our presentation is a discussion of the development of Rudimentary Linear Types, which are based on the $\otimes$, &, $\multimap$ fragment of linear logic. Both [Abr93] and [Gir89] give excellent introductions to linear logic, so the intent here is not to attempt to replicate their work. Instead, some of the standard intuitions behind linear logic, and the connectives that are part of Rudimentary Types are sketched.

2.1 Syntax and Typing Rules

Rudimentary Types, denoted by types θ, are built up using the following grammar:

$$\theta \quad ::= \quad \varphi \mid I \mid \theta \otimes \theta \mid \langle \theta, \theta \rangle \mid \theta \multimap \theta$$

Here, φ ranges over some unspecified set of primitive types. $\times$ (which is denoted by $\langle , \rangle$) and $\otimes$ (pronounced "tensor") are product type constructors, and I is the unit of $\otimes$. There is no programming reason to include a unit for $\times$. Before giving the syntax of terms, some standard intuitive readings of linear types are sketched.

One way of understanding linearity is as follows: Data is consumed as it is passed as input to a function. The function produces data as output, which is itself either consumed as it is passed as input to another function or returned as some sort of result. It is important that the data not be allowed to be duplicated or discarded. As such, two of the important structural rules in intuitionistic logic, those of Contraction (which allows a resource to be duplicated) and Weakening (which allows a resource to be discarded), are not part of intuitionistic linear logic. On this reading, functions of type $\theta \multimap \theta'$ use their argument exactly once. An element of type $\theta \otimes \theta'$ is a pair of elements, both of which must be used whenever the pair is. Elements of the type $\theta \times \theta'$ are more difficult to describe. Such an element consists of pairs, the components of which can be selected by projection. Using such a pair amounts to using one of the components, but not both.

A different understanding of these linear types can be couched in terms of lazy and eager evaluation. On this reading, a function of type $\theta \multimap \theta'$ is a call-by-value function, $\theta \otimes \theta'$ is a type of strict pairs (since both components of the $\otimes$ pair must be used, evaluation is eager), and $\theta \times \theta'$ is a type of lazy pairs (since only one component of the $\times$ pair will be used, via a projection, evaluation is lazy). This understanding is captured well by the standard cpo model, where $\multimap$ is the strict function space, $\otimes$ is the smash product, and $\times$ is the categorical product of cpo's.

One note of warning: it is dangerous to subscribe to these intuitions too seriously. First, it is nontrivial to formally categorize what these intuitions mean, especially the intuitions about consumption. Second, constructs that go against some of these intuitions will be shown later. Yet, these constructs remain perfectly well behaved within the defined mathematical framework and continue to satisfy the precise criteria that we previously set. These intuitions should serve only as a guide towards thinking about the problems to be presented.

Pre-terms M, N of the language of Rudimentary Types are built as follows (where k ranges over constants and x, y over variables):

$$\begin{array}{rcl} M, N & ::= & x \mid k \mid \mathbf{let}\ x \otimes y\ \mathbf{be}\ M\ \mathbf{in}\ N \mid MN \mid M \otimes N \mid \pi_1 M \mid \pi_2 M \mid \langle M, N \rangle \\ & & \mid \lambda x.M \mid \star \mid \mathbf{let}\ \star\ \mathbf{be}\ M\ \mathbf{in}\ N. \end{array}$$

A typing context Δ is a set of pairs of the form $x{:}\,\theta$ in which no variable x appears twice. This proviso must be respected in the use of comma for the disjoint union of contexts. (This proviso means that Δ_1, Δ_2 is defined only if the variables in Δ_1 and

Δ_2 are disjoint.) Since sets are used, there is no need for an Exchange rule. Every constant k comes with an associated type, $type(k)$. A judgement $\Delta \vdash M\colon \theta$ can be read as "In typing context Δ, term M has type θ".

$$\frac{}{x\colon \theta \vdash x\colon \theta}\ \text{Axiom} \qquad\qquad \frac{}{- \vdash k\colon type(k)}\ \text{Constant}$$

$$\frac{}{- \vdash \star\colon I}\ \star I \qquad\qquad \frac{\Delta_1 \vdash t\colon I \quad \Delta_2 \vdash u\colon \theta}{\Delta_1, \Delta_2 \vdash \mathbf{let}\,\star\ \mathbf{be}\ t\ \mathbf{in}\ u\colon \theta}\ \star E$$

$$\frac{\Delta \vdash P\colon \theta_1 \quad \Delta \vdash Q\colon \theta_2}{\Delta \vdash \langle P, Q\rangle\colon \theta_1 \times \theta_2}\ \times I \qquad\qquad \frac{\Delta \vdash P\colon \theta_1 \times \theta_2}{\Delta \vdash \pi_i\, P\colon \theta_i}\ \times E_i\ (i = 1, 2)$$

$$\frac{\Delta_1 \vdash P\colon \theta_1 \quad \Delta_2 \vdash Q\colon \theta_2}{\Delta_1, \Delta_2 \vdash P \otimes Q\colon \theta_1 \otimes \theta_2}\ \otimes I \qquad \frac{\Delta_1, x_1\colon \theta_1, x_2\colon \theta_2 \vdash t\colon \theta_3 \quad \Delta_2 \vdash u\colon \theta_1 \otimes \theta_2}{\Delta_1, \Delta_2 \vdash \mathbf{let}\ x_1 \otimes x_2\colon \theta_1 \otimes \theta_2\ \mathbf{be}\ u\ \mathbf{in}\ t\colon \theta_3}\ \otimes E$$

$$\frac{\Delta, \iota\colon \theta' \vdash P\colon \theta}{\Delta \vdash \lambda\iota\colon \theta'.P\colon \theta' \multimap \theta}\ \multimap I \qquad\qquad \frac{\Delta_1 \vdash P\colon \theta' \multimap \theta \quad \Delta_2 \vdash Q\colon \theta'}{\Delta_1, \Delta_2 \vdash PQ\colon \theta}\ \multimap E$$

Figure 1: Typing rules for Rudimentary Types

Figure 1 consists of all of the typing rules for Rudimentary Types. Note that the judgement $- \vdash M\colon \theta$ indicates that the typing context is empty. Every well-formed term must have exactly one rule that could have generated it. It is worthwhile to comment on the two rules for products. In the rule for $\times$I, since only one of the components of the pair will be selected via a projection, it is necessary that the typing context that makes up each of the two components of the pair be identical. In the rule for $\otimes$I, since each component of the pair *must* be used, the union of the typing contexts is made. The rules for Weakening and Contraction

$$\frac{\Delta \vdash M\colon \theta}{\Delta, x\colon \theta' \vdash M\colon \theta}\ \textsf{Weakening} \qquad \frac{\Delta, x\colon \theta', y\colon \theta' \vdash M\colon \theta}{\Delta, z\colon \theta' \vdash M[z/x, z/y]\colon \theta}\ \textsf{Contraction}$$

are simply not part of, and are not derivable in, Rudimentary Types. Let us illustrate the effect of this by revisiting the snapback example discussed in Chapter 1:

$$\mathit{snapback} \equiv \lambda s{:}\,\varphi.\lambda c{:}\,(\varphi \multimap (\varphi \otimes \mathbf{nat})).\mathbf{let}\ s' \otimes n\ \mathbf{be}\ cs\ \mathbf{in}\ s \otimes n.$$

In particular, we show that this term is not derivable, working backward from the derivation of the *snapback* function. The last step of the derivation must have been the result of a $\multimap$ I:

$$\frac{\Delta, s{:}\,\varphi \vdash \lambda c{:}\,(\varphi \multimap \varphi \otimes \mathbf{nat}).\mathbf{let}\ s' \otimes n\ \mathbf{be}\ cs\ \mathbf{in}\ s \otimes n{:}\,(\varphi \multimap \varphi \otimes \mathbf{nat}) \multimap (\varphi \otimes \mathbf{nat})}{\Delta \vdash \lambda s{:}\,\varphi.\lambda c{:}\,(\varphi \multimap \varphi \otimes \mathbf{nat}).\mathbf{let}\ s' \otimes n\ \mathbf{be}\ cs\ \mathbf{in}\ s \otimes n{:}\,\varphi \multimap (\varphi \multimap \varphi \otimes \mathbf{nat}) \multimap (\varphi \otimes \mathbf{nat})}$$

Working backwards, the previous step must have been the result of the following $\multimap$ I:

$$\frac{\Delta, s{:}\,\varphi, c{:}\,\varphi \multimap \varphi \otimes \mathbf{nat} \vdash \mathbf{let}\ s' \otimes n\ \mathbf{be}\ cs\ \mathbf{in}\ s \otimes n{:}\,\varphi \otimes \mathbf{nat}}{\Delta, s{:}\,\varphi \vdash \lambda c{:}\,(\varphi \multimap \varphi \otimes \mathbf{nat}).\mathbf{let}\ s' \otimes n\ \mathbf{be}\ cs\ \mathbf{in}\ s \otimes n{:}\,(\varphi \multimap \varphi \otimes \mathbf{nat}) \multimap (\varphi \otimes \mathbf{nat})}$$

And the next previous step must have been the result of the following $\otimes$ E:

$$\frac{s{:}\,\varphi, c{:}\,\varphi \multimap \varphi \otimes \mathbf{nat} \vdash cs{:}\,\varphi \otimes \mathbf{nat} \quad \Delta, s'{:}\,\varphi, n{:}\,\mathbf{nat} \vdash s \otimes n{:}\,\varphi \otimes \mathbf{nat}}{\Delta, s{:}\,\varphi, c{:}\,\varphi \multimap \varphi \otimes \mathbf{nat} \vdash \mathbf{let}\ s' \otimes n\ \mathbf{be}\ cs\ \mathbf{in}\ s \otimes n{:}\,\varphi \otimes \mathbf{nat}}$$

But the second clause in the premise is impossible to type check. It has an assumption, $s'{:}\,\varphi$, which is in the set of variables that make up the assumption but is not in the conclusion (Weakening is not allowed). Also, $s{:}\,\varphi$ must be in Δ, and this goes against the requirement of the $\otimes$E rule that the typing contexts in the precondition be disjoint (Contraction is not allowed). The reason $s{:}\,\varphi$ must be in Δ is because the formation of the term $s \otimes n{:}\,\varphi \otimes \mathbf{nat}$ must have been the result of an $\otimes$I, which would have required $\Delta \vdash s{:}\,\varphi$. The formation of this term must have been the result of the application of the Axiom, which requires that $\Delta = s{:}\,\varphi$. So, this *snapback* is not typable.

Note, however, that the *snapback* example *is* typable if we allowed Contraction and Weakening. In particular, instead of $\otimes$E, a Contraction could have been performed as follows:

$$\frac{\Delta, s_1{:}\,\varphi, s_2{:}\,\varphi, c{:}\,\varphi \multimap \varphi \otimes \mathbf{nat} \vdash \mathbf{let}\ s' \otimes n\ \mathbf{be}\ cs_1\ \mathbf{in}\ s_2 \otimes n{:}\,\varphi \otimes \mathbf{nat}}{\Delta, s{:}\,\varphi, c{:}\,\varphi \multimap \varphi \otimes \mathbf{nat} \vdash \mathbf{let}\ s' \otimes n\ \mathbf{be}\ cs\ \mathbf{in}\ s \otimes n{:}\,\varphi \otimes \mathbf{nat}}$$

Then, the $\otimes$E rule could have been applied as follows:

$$\frac{s_1\colon\varphi, c\colon\varphi \multimap \varphi\otimes\mathbf{nat} \vdash cs_1\colon\varphi\otimes\mathbf{nat} \quad \Delta, s_2\colon\varphi, s'\colon\varphi, n\colon\mathbf{nat} \vdash s_2\otimes n\colon\varphi\otimes\mathbf{nat}}{\Delta, s_1\colon\varphi, s_2\colon\varphi, c\colon\varphi\multimap\varphi\otimes\mathbf{nat} \vdash \mathbf{let}\ s'\otimes n\ \mathbf{be}\ cs_1\ \mathbf{in}\ s_2\otimes n\colon\varphi\otimes\mathbf{nat}}$$

This first premise is simply two occurrences of the Axiom rule followed by a $\multimap$ I rule followed by a $\multimap$ E rule firing. For the second premise, if Weakening were a valid rule, the following Weakening rule could have fired:

$$\frac{\Delta, s_2\colon\varphi, n\colon\mathbf{nat} \vdash s_2\otimes n\colon\varphi\otimes\mathbf{nat}}{\Delta, s_2\colon\varphi, s'\colon\varphi, n\colon\mathbf{nat} \vdash s_2\otimes n\colon\varphi\otimes\mathbf{nat}}$$

Now, this premise could have been obtained by two occurrences of the Axiom rule, followed by a $\otimes$I rule firing. As a result, (with the addition of Contraction and Weakening), *snapback* would be typable.

2.2 Properties of Linear Typing

Some basic properties of linear types are stated here. The proofs for these properties are all standard. These properties will be used later, primarily in Chapters 3 and 4, for proving properties of the functional and imperative operational semantics.

Proposition 1 (Free Variables I) *If* $\Delta \vdash M\colon\theta$, *then* $fv(M) = dom(\Delta)$.

Previously, we informally described linear variables as those that were "used once". The following definition of uniqueness is a formal criterion motivated by that informal description.

Definition 2 (Uniqueness) *The identifier x is said to be unique in M if it appears once, or if it appears uniquely within each arm of a pair $\langle M_1, M_2\rangle$ and nowhere else.*

The point of linear typing is that all variables are "unique" in the following sense:

Proposition 3 (Free Variables II) *If* $\Delta, x\colon\theta \vdash M\colon\theta'$, *$x$ is free uniquely in M.*

The next proposition, cut elimination, will be needed in Chapters 3 and 4 for proving subject reduction, and also in showing adequacy between the two operational semantics.

Proposition 4 (Cut Elimination) *If* $\Delta_1, x{:}\,\theta \vdash M{:}\,\theta'$ *and* $\Delta_2 \vdash N{:}\,\theta$ *(and* $x \notin \Delta_2$*), then*
$\Delta_1, \Delta_2 \vdash M[N/x]{:}\,\theta'$.

The proof of this proposition is by a straightforward and standard structural induction on the different possible forms of the term.

2.3 A Sample Language for Rudimentary Types

Our sample language has two primitive types:

$$\varphi \quad ::= \quad \mathbf{cell} \mid \mathbf{nat}$$

The type of storage cells is **cell**, and the type of natural numbers is **nat**. There are several function constants in the language. Here, the intuitive semantics of these function constants are sketched. They are formally defined (via their operational semantics) in Chapters 3 and 4. First are typical arithmetic operations ($+$ and $\dot{-}$), together with operations for explicitly copying and discarding natural numbers (these are needed because of linear typing). Natural numbers may be discarded and copied as they are simply constants. The conditional *ifz* tests its first argument for zero, selecting the first projection of the second argument if the test succeeds and the second projection otherwise.

$$\begin{array}{rcl@{\qquad}rcl}
+ & : & \mathbf{nat} \otimes \mathbf{nat} \multimap \mathbf{nat} & \dot{-} & : & \mathbf{nat} \otimes \mathbf{nat} \multimap \mathbf{nat} \\
\mathit{copy} & : & \mathbf{nat} \multimap \mathbf{nat} \otimes \mathbf{nat} & \mathit{discard} & : & \mathbf{nat} \multimap I \\
\mathit{ifz} & : & (\mathbf{nat} \otimes \langle \varphi, \varphi \rangle) \multimap \varphi & n & : & \mathbf{nat}
\end{array}$$

Next are a number of function constants for creating, overwriting, reading, and removing cells:

$$\begin{array}{rclcrcl} \mathit{alloc} & : & \mathbf{nat} \multimap \mathbf{cell} & \qquad & \mathit{dealloc} & : & \mathbf{cell} \multimap I \\[1em] \mathit{lookup} & : & \mathbf{cell} \multimap \mathbf{cell} \otimes \mathbf{nat} & & \mathit{update} & : & \mathbf{nat} \otimes \mathbf{cell} \multimap \mathbf{cell} \end{array}$$

The function call $alloc(n)$ creates a storage cell with initial value n, and $dealloc(c)$ deallocates the cell, c. The function *lookup* returns the value in the cell, along with the cell itself. (This function *must* return the single pointer to the cell, because of linearity.) There are two ways to interpret the *update* constant. The first is a functional way: performing an *update* will result in a new cell being returned, with the value as specified in the other argument to *update*. In the second, an imperative way, an explicit store is used; as such, when an *update* is performed, the actual cell is changed. In later chapters, coincidence between these two interpretations will be shown.

Note that *dealloc* could be given the type $\mathbf{cell} \multimap \mathbf{nat}$ which returns the value in the cell before discarding it—the two versions are programmable in terms of one another. Finally, it would be trivial to consider extensions such as multi-element arrays, multi-dimensional arrays, or other related data structures. The consideration of single cells suffices to make the main points in this work and simplifies the presentation. (A later chapter will use multi-element and multi-dimensional arrays).

2.4 Repeat

So far, the linear language is limited in many ways, most important in its lack of a provision for recursion. A limited form of recursion can be provided by adding a function constant for indefinite iteration:

$$\mathbf{repeat} \colon (\mathbf{cell} \multimap \mathbf{nat} \otimes \mathbf{cell}) \multimap (\mathbf{cell} \multimap \mathbf{cell}).$$

The uncurried form of the function **repeat** takes two arguments, a function (which takes a **cell** as an argument and returns a **nat** and a **cell**) and a **cell**. Intuitively, the function **repeat** repeatedly applies the function of type $\mathbf{cell} \multimap \mathbf{nat} \otimes \mathbf{cell}$ to the argument of type **cell** until a zero is produced in the output position (the **nat** in the $\mathbf{nat} \otimes \mathbf{cell}$ returned by the function). The operational semantics of **repeat** appear in the next two chapters. This constant allows for a significant amount of Fortran-style programming. The most severe limitation is that it is not possible to program recursively defined functions, especially ones with multiple recursive calls. This limitation is part of the motivation for considering extensions to rudimentary types in later chapters.

Notice that this **repeat** combinator violates the intuition that functions of $\multimap$ type use their arguments exactly once. (This failure of intuitive notions is one of the reasons we look for precise mathematical criteria as a means of describing properties of the language of Rudimentary Types and its extensions.) The previous intuitions about "used once" are *not* necessary for understanding cells in the imperative way. It will be shown that the **repeat** construct does not break the coincidence between the functional and imperative operational semantics.

Chapter 3

Functional Semantics and Properties

In this chapter, we develop a functional operational semantics for the language of Rudimentary Types. The purpose of doing this is twofold. First, we obtain a clear reference semantics; that is, we develop a semantics to which other operational semantics may be compared. Second, we emphasize the functional character of the language of Rudimentary Types.

In regards to the functional character of the language, there are three important points:

1. Reductions are allowed to occur in any order.

2. Adequacy is shown with respect to a natural denotational model that is formulated using values and functions, but with *no* states.

3. An operational extensionality theorem is then proved.

These are all properties expected to hold in a functional language. We consider these properties as precise criteria indicating the functional character of the language.

The chapter is broken up into two parts. In the first, we define the functional operational reduction rules for Rudimentary Types, in which reductions may occur in any order. In the second part of the chapter, we formulate two main results: adequacy with respect to an appropriately defined denotational model of the language (based on $\mathbf{CPO}_\perp$ with strict maps) and an operational extensionality theorem.

3.1 Operational Semantics

3.1.1 Definitions

Figure 2 contains the (standard) reduction rules for Rudimentary Types that are the same for both the functional and the imperative operational semantics. Here, $\underline{n}$ refers to the actual constant corresponding to the natural number n. This terminology is necessary to separate the $+$ and $-$ operators within the language from arithmetic addition and subtraction.

$$
\begin{array}{rcll}
(\lambda x.M)N & \rightarrow & M[N/x] & \\
\mathbf{let}\ x \otimes y\ \mathbf{be}\ M \otimes N\ \mathbf{in}\ C & \rightarrow & C[M/x, N/y] & \\
\pi_i\langle M_1, M_2\rangle & \rightarrow & M_i & (\text{for } i = 1, 2) \\
\mathbf{let}\ \star\ \mathbf{be}\ \star\ \mathbf{in}\ C & \rightarrow & C & \\
\mathit{ifz}(\underline{0} \otimes M) & \rightarrow & \pi_1 M & \\
\mathit{ifz}(\underline{n} \otimes M) & \rightarrow & \pi_2 M & (\text{if } \underline{n} \neq \underline{0}) \\
+(\underline{n} \otimes \underline{m}) & \rightarrow & \underline{n+m} & \\
\dot{-}\ (\underline{n} \otimes \underline{m}) & \rightarrow & \underline{n-m} & (\text{if } m < n) \\
\dot{-}\ (\underline{n} \otimes \underline{m}) & \rightarrow & 0 & (\text{if } m \geq n)
\end{array}
$$

Figure 2: Standard reductions for operational semantics for Rudimentary Types

$$\begin{array}{lcl}
update(\underline{n},\ alloc(\underline{m})) & \rightarrow_F & alloc(\underline{n}) \\
lookup(alloc(\underline{n})) & \rightarrow_F & \underline{n} \otimes alloc(\underline{n}) \\
dealloc(alloc(\underline{n})) & \rightarrow_F & \star
\end{array}$$

$$\frac{M \rightarrow_F N}{C[M] \rightarrow_F C[N]}$$

$$\frac{M \rightarrow N}{M \rightarrow_F N}$$

$$\begin{array}{lcl}
\textbf{repeat}\ f\ l & \rightarrow_F^{(k+1)} & l' \quad (\text{if } f\ l \rightarrow_F^k \underline{0} \otimes l') \\
& \rightarrow_F^{(k+1)} & \textbf{repeat}\ f\ l' \quad (\text{if } f\ l \rightarrow_F^k \underline{n} \otimes l', \underline{n} \neq \underline{0})
\end{array}$$

Figure 3: Functional Operational Semantics

The functional operational semantics $\rightarrow_F$ extends $\rightarrow$ as specified in Figure 3. Note that $\rightarrow_F^k$ means "reduces in k steps", and that $\rightarrow_F$ is shorthand for $\rightarrow_F^1$. Also, the notation $\rightarrow_F^*$ is used to refer to a reduction of zero or more steps. We make an important comment about the meaning of "reduces in k steps". It does *not* mean that there are k single-step reductions. Rather, there is an implicit rule:

$$\frac{M \rightarrow_F^k N \quad N \rightarrow_F^j P}{M \rightarrow_F^{(k+j)} P}$$

This understanding is crucial for proving subject reduction, as otherwise (if all reductions had to be single-step) subject reduction involving nested **repeat** loops would not be provable.

The intuition behind these reductions is that *update* receives (as an argument) an allocated cell and a value, and returns an allocated cell set to the value passed in as the input. The *lookup* function takes in a cell and returns a copy of the cell tensored with its value; note that *lookup must* return the cell as the cell would otherwise be lost. Finally, *dealloc* of an *alloc*ed cell simply returns $\star$. These reduction rules will be different in the imperative operational semantics, as a state parameter will be used there to keep track of the cells. The one other rule for which it is worthwhile to comment is the **repeat** rule. The intuitive definition for **repeat** would have been to apply the function to the argument, and test the resulting value, evaluating **repeat** again if the value were non-zero. It could be represented as the following:

$$\begin{array}{rcl} \mathbf{repeat}\ f\ l & \to_F & \mathbf{let}\ x \otimes l'\ \mathbf{be}\ f\ l\ \mathbf{in} \\ & & \quad \mathit{ifz}\,(x \otimes \langle l', \mathbf{repeat}\ f\ l' \rangle) \end{array}$$

The problem with this definition is that subject reduction (as discussed below) would fail using this definition. This definition requires the use of both $\mathsf{Weakening}$ of the expression f in order to type $\langle l', \mathbf{repeat}\ f\ l' \rangle$, and $\mathsf{Contraction}$ of f to type the whole term. There is no requirement that f not have any free identifiers. (This is contrasted with the "correct" definition of **repeat** which requires that f *not* have any free identifiers as $f\ l$ reduces to $\underline{n} \otimes l'$ in k steps, and, because of uniqueness, Δ must be empty in order to provide a valid typing for $\Delta \vdash \underline{n} \otimes l{:}\, \mathbf{nat} \otimes \mathbf{cell}$.) And, since f is *not* of type **nat**, we may neither contract nor weaken it. As such, the slightly more complicated definition must be used.

3.1.2 Subject Reduction

Subject reduction is needed for the adequacy proof between the functional and imperative operational semantics in the next chapter, and for the adequacy between the functional operational semantics and the denotational model. It also implies that reductions may occur in any order.

Proposition 5 (Subject Reduction for functional operational semantics) *If* $\Delta \vdash M{:}\,\theta$ *and* $M \to_F^* N$*, then* $\Delta \vdash N{:}\,\theta$*.*

Proof: The proof is by an induction on the structure of the derivation of the judgement of $M \to_F^k N$. The proof is accomplished by a case analysis, considering each of the rules that could have been used in the reduction of the term M. Most of the reduction rules are standard, and have been proved elsewhere. The only interesting cases are those which involve the function constants for **cell**s and **repeat**. Even the cases for the array reductions are similar to the corresponding cases from Theorem 34, the corresponding Subject Reduction theorem for the imperative operational semantics. Note that the implicit multi-step rule mentioned in the previous section must also be considered, but it is trivial from the induction hypothesis.

Case: M is $update(\underline{n}, alloc(\underline{m}))$.

Thus, it is necessary to show that if $\Delta \vdash update(\underline{n}\, alloc(\underline{m}))\colon \mathbf{cell}$, then $\Delta \vdash alloc(\underline{n})\colon \mathbf{cell}$. It is first noted, that because of the fact that $\underline{n}$, $\underline{m}$, $alloc$, and $update$ are all constants, and by Propositions 3 from Section 2.2 concerning uniqueness and Proposition 1 from the same section, that $\Delta = \{\}$. Since $alloc$ is just a constant of type $\mathbf{nat} \multimap \mathbf{cell}$ and $\underline{n}$ is a constant of type $\mathbf{nat}$, by applying the rule for application,

$$- \vdash alloc(\underline{n})\colon \mathbf{cell}.$$

Case: M is $lookup(alloc(\underline{n}))$.

Then, it is necessary to show that if $\Delta \vdash lookup(alloc(\underline{n}))\colon \mathbf{cell} \otimes \mathbf{nat}$, then $\Delta \vdash alloc(\underline{n}) \otimes \underline{n}\colon \mathbf{cell} \otimes \mathbf{nat}$. Again, it is noted that $\Delta = \{\}$. From the previous case, $- \vdash alloc(\underline{n})\colon \mathbf{cell}$. Since $- \vdash \underline{n}\colon \mathbf{nat}$, the rule for $\otimes$-introduction may be applied,

$$\frac{- \vdash \underline{n}\colon \mathbf{nat} \quad - \vdash alloc(\underline{n})\colon \mathbf{cell}}{- \vdash alloc(\underline{n}) \otimes \underline{n}\colon \mathbf{cell} \otimes \mathbf{nat}}$$

and the desired result is obtained.

Case: M is $dealloc(alloc(\underline{n}))$.

It is necessary to show that if $\Delta \vdash dealloc(alloc(\underline{n}))\colon I$, then $\Delta \vdash \star\colon I$. Again, it is noted that $\Delta = \{\}$. Since $\star$ is just a constant, $- \vdash \star\colon I$, and there is nothing to prove.

Case: M is **repeat** $f\ \ell$.

There are two possible subcases:

Subcase: M is reduced by

$$\frac{(f\ \ell) \ \rightarrow_F^k \ \underline{n} \otimes \ell'}{\mathbf{repeat}\ f\ \ell \ \rightarrow_F^{(k+1)} \ \mathbf{repeat}\ f\ \ell'}\ n <> 0$$

Here, it is necessary to show that $\Delta \vdash \textbf{repeat}\ f\ \ell' \colon \textbf{cell}$ if $\Delta \vdash \textbf{repeat}\ f\ \ell \colon \textbf{cell}$. The term $\textbf{repeat}\ f\ \ell \colon \textbf{cell}$ must have been the result of the following function application:

$$\frac{\Delta \vdash \textbf{repeat}\ f \colon \textbf{cell} \multimap \textbf{cell} \quad \Delta' \vdash \ell \colon \textbf{cell}}{\Delta, \Delta' \vdash \textbf{repeat}\ f\ \ell \colon \textbf{cell}}$$

Due to uniqueness (Proposition 3 from Chapter 2), $\Delta' = \{\}$. The function application may be rewritten as:

$$\frac{\Delta \vdash \textbf{repeat}\ f \colon \textbf{cell} \multimap \textbf{cell} \quad - \vdash \ell \colon \textbf{cell}}{\Delta \vdash \textbf{repeat}\ f\ \ell \colon \textbf{cell}}$$

But, since $- \vdash \ell' \colon \textbf{cell}$, it is possible to immediately apply the following function application to obtain the desired result:

$$\frac{\Delta \vdash \textbf{repeat}\ f \colon \textbf{cell} \multimap \textbf{cell} \quad - \vdash \ell' \colon \textbf{cell}}{\Delta \vdash \textbf{repeat}\ f\ \ell' \colon \textbf{cell}}$$

The other **repeat** case is slightly different, and takes advantage of the induction hypothesis.

Subcase: M is reduced by

$$\frac{f\ \ell\ \rightarrow_F^k\ \underline{0} \otimes \ell'}{\textbf{repeat}\ f\ \ell\ \rightarrow_F^{(k+1)}\ \ell'}$$

Here, it is necessary to show that $\Delta \vdash \ell' \colon \textbf{cell}$ if $\Delta \vdash \textbf{repeat}\ f\ \ell \colon \textbf{cell}$.

Since the structure of the judgement of $f\ \ell\ \rightarrow_F^k\ \underline{0} \otimes \ell'$ is "simpler" (it is the precedent in the typing judgement for **repeat**), by the induction hypothesis we have that $\Delta \vdash \underline{0} \otimes \ell' \colon \textbf{nat} \otimes \textbf{cell}$ when $\Delta \vdash f\ \ell \colon \textbf{nat} \otimes \textbf{cell}$. By uniqueness, we know that $\Delta = \{\}$. Now the term $\underline{0} \otimes \ell'$ may have been constructed as follows:

$$\frac{- \vdash \underline{0} \colon \textbf{nat} \quad - \vdash \ell' \colon \textbf{cell}}{- \vdash \underline{0} \otimes \ell' \colon \textbf{nat} \otimes \textbf{cell}}$$

Because of uniqueness, we have that: $- \vdash \ell' \colon \textbf{cell}$.

■

The next two sections cover the adequacy of this functional operational semantics with respect to a denotational semantics to be defined, and the proof of operational extensionality.

3.2 A Denotational Model

By a cpo, we mean an ω-complete poset possessing a least element. Before presenting the model, a few brief definitions need to be provided. The domain $N_\perp$ consists of the same elements of the domain N (the discreetly ordered natural numbers) together with a new element $\perp$ such that for all $n \in N$, $\perp \sqsubseteq n$. A strict continuous function f from $D_\perp$ to $E_\perp$ is a continuous function with the added restriction that $f(\perp_D) = \perp_E$. The cartesian product of two cpo's $D_\perp$ and $E_\perp$ is a domain $D_\perp \times E_\perp$ consisting of the elements $\{(d, e) \mid d \in D_\perp \wedge e \in E_\perp\}$ with the resulting partial order defined componentwise. The smash product of two cpo's $D_\perp$ and $E_\perp$ is a domain $D_\perp \otimes E_\perp$ consisting of elements $\{(d, e) \in D_\perp \times E_\perp \mid d \neq \perp_D \text{ and} e \neq \perp_E\} \cup \{\perp_{D_\perp \otimes E_\perp}\}$ having the componentwise ordering with the added restriction that for all $x \in D_\perp \otimes E_\perp$, $\perp_{D_\perp \otimes E_\perp} \sqsubseteq x$. The denotational model takes place in the category $\mathbf{CPO}_\perp$ of cpo's and strict continuous functions [Plo78].

The semantics of types can be given as follows, where N is the set of natural numbers.

$$[\![\mathbf{cell}]\!] = N_\perp$$

$$[\![\mathbf{nat}]\!] = N_\perp$$

$$[\![I]\!] = I, \text{the two point cpo}$$

$$[\![\theta \multimap \theta']\!] = \text{strict continuous functions from } [\![\theta]\!] \text{ to } [\![\theta']\!], \text{ pointwise ordered}$$

$$[\![\theta \times \theta']\!] = \text{the cartesian product of } [\![\theta]\!] \text{ and } [\![\theta']\!]$$

$$[\![\theta \otimes \theta']\!] = \text{the smash product of } [\![\theta]\!] \text{ and } [\![\theta']\!]$$

The main point is that no state parameter is needed in explicating the semantics, further underlining the functional character of the language. But there is also a strange point to note: **cell** and **nat** are isomorphic. It would have been possible, in fact, to dispense with **cell** altogether, and have a **nat** type with certain operations (such as successor) done destructively. However, the treatment of **cell** is quite natural, and conveniently extensible to arrays and other more complex data structures. Furthermore, a more meaningful distinction can be drawn when considering complexity, which is not accounted for in $\mathbf{CPO}_\perp$. The issue of complexity will be revisited in Chapter 7.

The four **cell** function constants may be interpreted by the following $\mathbf{CPO}_\perp$-maps:

$$\begin{array}{rcl} alloc & : & N_\perp \multimap N_\perp \\ update & : & N_\perp \otimes N_\perp \multimap N_\perp \\ lookup & : & N_\perp \multimap N_\perp \otimes N_\perp \\ dealloc & : & N_\perp \multimap I \end{array}$$

where *alloc* is the identity function; *update* is the first projection function; *lookup* is the contraction map (i.e., the strict extension of the *copy* map); and, *dealloc* is the map that sends the bottom element to the bottom element in the two point cpo, and sends all other elements to the non-bottom element in the two point cpo. In other words,

$$\begin{array}{rcl} [\![alloc]\!] & = & \mathrm{id}_{N_\perp} \\ [\![update]\!] & = & \{((n,m),n) \mid (n,m) \in N_\perp \otimes N_\perp\} \\ [\![lookup]\!] & = & \{(n, n \otimes n) \mid n \in N_\perp\} \\ [\![dealloc]\!] & = & \{(n, \top_I) \mid n \in N_\perp \wedge n \neq \perp_{N_\perp}\} \cup \{(\perp_{N_\perp}, \perp_I)\}. \end{array}$$

The semantic equations for other terms should be relatively obvious, so the connection of this model to the operational semantics is simply stated.

Theorem 6 (Adequacy) *If* $- \vdash M \colon \mathbf{nat}$*, then*

$$\forall n \in N \,.\, [\![M]\!] = n \iff M \rightarrow_F^* \underline{n}$$

Proof: [Sketch] Braüner in [Bra96] has verified adequacy results for linear λ-calculi with respect to categorical models, one of which is the model $\mathbf{CPO}_\perp$ that is used here. The main difference from our work is that he uses a many-step lazy operational semantics based on evaluation to "canonical form", where a judgement of the form $M \Downarrow V$ means that M evaluates to canonical form V. In contrast, this thesis uses a one-step semantics (except for **repeat**).

Rather than prove adequacy directly, which would involve replicating some of Braüner's work, a simple outline is provided for how the proof can be essentially derived from his results.

1. Braüner proves adequacy for a linear version of PCF, using a many-step semantics. His operational semantics and adequacy proof can be modified slightly (to allow for **cell**, which in the model is isomorphic to **nat**) to give an adequacy result for our language with respect to a semantics formulated in terms of a many-step relation $M \Downarrow V$.

2. A connection can be established between the many-step and one-step semantics to the effect that
$$M \Downarrow V \iff M \rightarrow^{*} V.$$
This can be accomplished as in [Plo75].

These two points immediately imply our adequacy theorem. Since their further development is routine and standard, no additional detail is provided. ■

An immediate consequence of the theorem is the determinacy of the operational semantics.

Corollary 7 (Determinacy) *If* $- \vdash M\colon \mathbf{nat}$*, then*

$$\forall \underline{n}, \underline{m} \in \mathbf{nat} \,.\, M \rightarrow_F^{*} \underline{n} \;\wedge\; M \rightarrow_F^{*} \underline{m} \;\Rightarrow\; \underline{n} = \underline{m}$$

Of course, determinacy could be proved directly by syntactic means without appeal to a denotational model.

We view this denotational model is as providing evidence for the "functional character" of the language for the following two reasons:

1. In the model, function types consist of functions, pair types consist of set theoretic pairs, etc.

2. There are no states in the model to explain *alloc*, *dealloc*, *lookup*, and *update*.

A possible criticism of this view is that the model is not as accurate as one might like. For instance, it contains Contraction functions at all types, such as the evident map of type

$$(\mathbf{cell} \multimap \mathbf{cell}) \multimap ((\mathbf{cell} \multimap \mathbf{cell}) \otimes (\mathbf{cell} \multimap \mathbf{cell})).$$

The next section contains a response to this criticism — there the functional characteristics of the language are strengthened. An operational extensionality result is given. This result is based intuitively on the denotational model, but is formulated in terms of contextual equivalence and approximation.

3.3 Operational Extensionality

Development of an idea of operational extensionality is desirable because we want to formalize the notion that function types in our programming language correspond to mathematical functions. In particular, we wish to develop an analogous result to the context lemma in PCF [Mil77]. However, when formulating the definition of operational extensionality (in terms of imposing an ordering on syntactic terms), there is a problem when trying to interpret $\otimes$. The naive approach to attempting to define order for $\otimes$ as, say,

$$M_1 \otimes M_2 \sqsubseteq N_1 \otimes N_2 \iff M_1 \sqsubseteq N_1 \wedge M_2 \sqsubseteq N_2$$

does not work, because there are terms of tensor type which are not of the form $M_1 \otimes M_2$. For example, in the *snapback* example from Chapter 2, the function c has type $\varphi \multimap \varphi \otimes \mathbf{nat}$ and the argument s has type φ. The result of the application, cs, has type $\varphi \otimes \mathbf{nat}$. But while this is perfectly well typed, the term cs is not of the form $M_1 \otimes M_2$. As such, we need to use $\otimes$'s domain theoretic counterpart for deciding how one term can $\otimes$-approximate another.

Observational approximation is defined as follows.

Definition 8 *$C[\cdot]$ is a $\Gamma\theta$-**context** if $- \vdash C[M] : \mathbf{nat}$ whenever $\Gamma \vdash M : \theta$.*

Definition 9 *For $\Gamma \vdash M : \theta$ and $\Gamma \vdash N : \theta$, we say that "M observationally approximates N", written as $\Gamma \vdash M \sqsubseteq N$, if and only if $\forall\, \Gamma\theta$-contexts $C[\cdot]\,.\,\forall \underline{n}\,.\,C[M] \rightarrow_F^* \underline{n} \Rightarrow C[N] \rightarrow_F^* \underline{n}$.*

We now state an operational extensionality result.

Theorem 10 (Operational Extensionality) *Suppose that $- \vdash M : \theta$ and $- \vdash N : \theta$. Then, the following properties hold:*

- *If $\theta = \mathbf{nat}$, then $- \vdash M \sqsubseteq N$ iff for all constants $\underline{n}$: $\mathbf{nat}$,*

$$M \rightarrow_F^* \underline{n} \implies N \rightarrow_F^* \underline{n}.$$

- *If $\theta = \mathbf{cell}$, then $- \vdash M \sqsubseteq N$ iff for all constants $\underline{n}$: $\mathbf{nat}$,*

$$M \rightarrow_F^* alloc(\underline{n}) \implies N \rightarrow_F^* alloc(\underline{n}).$$

- *If $\theta = \theta_1 \multimap \theta_2$, then $- \vdash M \sqsubseteq N$ iff for all terms $V : \theta_1$,*

$$- \vdash MV \sqsubseteq NV.$$

- *If $\theta = \langle \theta_1, \theta_2 \rangle$, then $- \vdash M \sqsubseteq N$ iff*

$$- \vdash \pi_1 M \sqsubseteq \pi_1 N \quad \wedge \quad - \vdash \pi_2 M \sqsubseteq \pi_2 N.$$

- *If $\theta = \theta_1 \otimes \theta_2$, then $- \vdash M \sqsubseteq N$ iff*

$$(\not\exists P, Q. M \rightarrow_F^* P \otimes Q) \vee (\exists P, Q. M \rightarrow_F^* P \otimes Q \wedge (P \sqsubseteq \Omega_{\theta_1} \vee Q \sqsubseteq \Omega_{\theta_2} \vee$$

$$(\exists P', Q' \ni N \rightarrow_F^* P' \otimes Q' \wedge - \vdash P \sqsubseteq P' \wedge - \vdash Q \sqsubseteq Q'))).$$

The proof of this theorem appears in the next section. We will occasionally use the abbreviated notation $M \sqsubseteq N$ to mean $- \vdash M \sqsubseteq N$.

Here $\Omega_{\mathbf{cell}}$ can be defined to be the program:

$$\mathbf{repeat}(\lambda x.lookup(x))(alloc(1))$$

(i.e., a repeat loop which never terminates). $\Omega_{\mathbf{nat}}$ can be defined to be the program:

$$\mathbf{let}\ val \otimes l\ \mathbf{be}\ \Omega_{\mathbf{cell}}\ \mathbf{in}\ \ \mathbf{let}\ \star\ \mathbf{be}\ dealloc(l)\ \mathbf{in}\ val$$

$\Omega_{\mathbf{nat} \multimap \mathbf{nat}}$ can be defined to be the program, $\lambda x{:}\,\mathbf{nat}.\Omega_{\mathbf{nat}}$. And, Ω_θ for other types is similarly defined.

It is useful to explain the case for $\otimes$. In this case, $- \vdash M \sqsubseteq N$ if one of the following three conditions holds:

1. M does not reduce to a tensor pair.
2. M reduces to a tensor pair, but one of the pair is equivalent to Ω.
3. M reduces to a tensor pair $P \otimes Q$, N reduces to a tensor pair $P' \otimes Q'$, and $P \sqsubseteq P'$ and $Q \sqsubseteq Q'$.

This case is motivated by the definition of smash product in domain theory. Recall that, for cpo's A and B,

$$A \otimes B \ \equiv\ \{(a, b) \in A \times B \mid a = \bot \ \leftrightarrow\ b = \bot\}, \text{ ordered componentwise.}$$

Point 3 above corresponds to the componentwise order, while point 2 corresponds to the fact that if one of the components is $\bot$ then so is the pair. Point 1 covers the case of terms that do not reduce to tensor pairs, which are equivalent to $\bot$.

This theorem shows a sense in which function types may be regarded as ordered sets of mathematical functions. These functions take equivalence classes of closed terms as arguments and produce equivalence classes as results. These equivalence classes are obtained from observational equivalence, defined by

$$M \equiv N \quad \Longleftrightarrow \quad M \sqsubseteq N \ \wedge\ N \sqsubseteq M$$

Additionally, as we have noted, $\otimes$ corresponds closely to the smash product in $\mathbf{CPO}_\perp$, and $\times$ consists of pairs ordered like the cartesian product in $\mathbf{CPO}_\perp$. Thus, we regard operational extensionality as a syntactic result that accentuates the functional character of the language, strengthening and reiterating the view given by the model in $\mathbf{CPO}_\perp$.

3.3.1 Proof of Operational Extensionality

The proof of operational extensionality is based on the technique of applicative simulations. This technique uses, as its basis, an appropriately defined binary relation. The basic proof method is covered in great detail by Pitts in [Pit95] and [Pit94]; another example of this technique may be seen in [Bie97]. Below, the major points of Pitts' proof (which uses a method developed by Howe) are given for the functional operational semantics. This is a general method which we have not tried to simplify for our language. We simply give the details of the proof. Some readers may wish to skip to the next section.

The following rules can be used to help define the notion of a pre-congruence $\mathcal{R}$ for rudimentary types:

Definition 11 (pre-congruence for rudimentary types) *A family of relations $\mathcal{R}_{\Delta,\theta} \subseteq Exp_\Delta(\theta) \times Exp_\Delta(\theta)$ is a pre-congruence for rudimentary types if it satisfies the rules in Figure 4.*

Proposition 12 $\sqsubseteq$ *is a pre-congruence.*

Proof: By inspection. ■

Since it is difficult to work with observational equivalence as it is defined ($\forall\Gamma\theta$ contexts are difficult to handle), the process of applicative simulation is instead used (and it may then be compared to observational equivalence). For any binary relation $\mathcal{S} \subseteq Prog \times Prog$ (where $Prog$ is defined to be the set of closed terms), let $[\mathcal{S}]$ be defined as:

$$\frac{\Delta \vdash M\colon \theta}{\Delta \vdash M \mathcal{R} M\colon \theta}\ \text{Reflexivity}$$

$$\frac{\Delta \vdash M \mathcal{R} N\colon \theta \quad \Delta \vdash N \mathcal{R} P\colon \theta}{\Delta \vdash M \mathcal{R} P\colon \theta}\ \text{Transitivity}$$

$$\frac{\Gamma \vdash M\colon \theta_2 \quad \Delta, x\colon \theta_2 \vdash N \mathcal{R} P\colon \theta_1}{\Gamma, \Delta \vdash N[M/x] \mathcal{R} P[M/x]\colon \theta_1}\ Subs_1$$

$$\frac{\Gamma \vdash M \mathcal{R} N\colon \theta_2 \quad \Delta, x\colon \theta_2 \vdash P\colon \theta_1}{\Gamma, \Delta \vdash P[M/x] \mathcal{R} P[N/x]\colon \theta_1}\ Subs_2$$

$$\frac{\Gamma \vdash M \mathcal{R} M'\colon \theta_1 \otimes \theta_2 \quad \Delta, x\colon \theta_1, y\colon \theta_2 \vdash N \mathcal{R} N'\colon \theta_3}{\Gamma, \Delta \vdash \mathbf{let}\ M\ \mathbf{be}\ x \otimes y\ \mathbf{in}\ N \mathcal{R} \mathbf{let}\ M'\ \mathbf{be}\ x \otimes y\ \mathbf{in}\ N'\colon \theta_3}\ \otimes - E$$

$$\frac{\Delta \vdash M \mathcal{R} M'\colon \theta_1 \times \theta_2}{\Delta \vdash \pi_i(M) \mathcal{R} \pi_i(M')\colon \theta_i}\ \times - E$$

Figure 4: Pre-congruence relations for Rudimentary Types

$$P[\mathcal{S}]P' \equiv$$

$$\begin{array}{ll}
\forall \underline{n} \in \mathbf{nat} & P \rightarrow_F^* \underline{n} \Rightarrow P' \rightarrow_F^* \underline{n} \\
\forall \underline{n} \in \mathbf{nat} & P \rightarrow_F^* alloc(\underline{n}) \Rightarrow P' \rightarrow_F^* alloc(\underline{n}) \\
\forall P_1, P_2 \colon \theta_1 \times \theta_2 & \pi_1(P_1)\mathcal{S}\pi_1(P_1') \ \wedge \ \pi_2(P_2)\mathcal{S}\pi_2(P_2') \\
\forall P_1, P_2 \colon \theta_1 \otimes \theta_2 & (\not\exists S, T \ni P \rightarrow_F^* S \otimes T) \vee \\
 & (\exists S, T \ni P_1 \rightarrow_F^* S \otimes T \wedge (S \equiv \Omega_{\theta_1} \vee T \equiv \Omega_{\theta_2} \vee \\
 & (\exists S', T' \ni P_2 \rightarrow_F^* S' \otimes T' \ \wedge S\mathcal{S}S' \ \wedge \ T\mathcal{S}T'))) \\
\forall P_1, P_2 \colon \theta \multimap \theta' & \forall Q \colon \theta \ \ P_1 Q \mathcal{S} P_2 Q
\end{array}$$

Proposition 13 *The function $S \mapsto [S]$ is monotone.*

Since this function is monotone, and since the families of relations (as indexed by their types) form a complete lattice, the function has a greatest fixed point, $\leq$. This greatest fixed point is known as an applicative simularity. And by co-induction, to show that $P \leq P'$, it is only necessary to find some applicative simulation, $\mathcal{S}$, where $P\mathcal{S}P'$. The relation $\leq$ can be extended to all expressions (not just closed ones) by the following: Given $\Gamma = \{x_1, \ldots, x_n\}$, and $M, N \in Exp(\Gamma)$,

$$\Gamma \vdash M \leq^o N \Leftrightarrow \forall P_1, \ldots, P_n \in Prog \ \ M[\overrightarrow{P}/\overrightarrow{x}] \leq N[\overrightarrow{P}/\overrightarrow{x}].$$

(Note that $\leq^o$ is used to refer to the $\leq$ relation for open terms.) So, $P \leq P'$ implies that $P \rightarrow_F^* \underline{n} \Rightarrow P' \rightarrow_F \underline{n}$. And, the overall theorem of operational extensionality can be rephrased as:

Theorem 14 (Operational Extensionality rephrased)

$$\Gamma \vdash M \leq M' \Leftrightarrow \Gamma \vdash M \sqsubseteq M'.$$

Proof: (of the $\Rightarrow$ direction). Given that $\Gamma \vdash M \leq M'$, it is necessary to show that $\Gamma \vdash M \sqsubseteq M'$.

Lemma 15 contains some basic properties of $\leq$. The overall technique of the proof in this direction is to show that $\leq$ is a pre-congruence and to combine this with Lemma 15 to obtain the desired result.

Lemma 15 (Congruence of $\leq$) $\leq$ *has the following congruence properties:*

$$
\begin{array}{lll}
\Gamma \vdash M \leq M' & \Rightarrow & \Gamma \vdash (op\ M) \leq (op\ M') \\
\Gamma \vdash M \leq M' \wedge \Gamma \vdash N \leq N' & \Rightarrow & \Gamma \vdash (op\ M\ N) \leq (op\ M'\ N') \\
\Gamma \vdash M \leq M' \wedge \Gamma \vdash N \leq N' & \Rightarrow & \Gamma \vdash \langle M, N\rangle \leq \langle M', N\rangle \\
\Gamma \vdash M \leq M' \wedge \Gamma \vdash N \leq N' & \Rightarrow & \Gamma \vdash M \otimes N \leq M' \otimes N' \\
\Gamma \vdash M \leq M' \wedge \Gamma \vdash N \leq N' & \Rightarrow & \Gamma \vdash \pi_i\langle M, N\rangle \leq \pi_i\langle M', N\rangle \\
\Gamma \vdash M \leq M' \wedge \Gamma \vdash N \leq N' \wedge \Gamma, x, y \vdash T \leq T' & \Rightarrow & \Gamma \vdash \textbf{let } M \otimes N \textbf{ be } x \otimes y \textbf{ in } T \leq \\
 & & \quad \textbf{let } M' \otimes N' \textbf{ be } x \otimes y \textbf{ in } T' \\
\Gamma, x{:}\,\theta \vdash M \leq M' & \Rightarrow & \Gamma \vdash \lambda x.M \leq \lambda x.M' \\
\Gamma \vdash M \leq M' \wedge \Gamma \vdash N \leq N' & \Rightarrow & \Gamma \vdash MN \leq M'N'
\end{array}
$$

Note: $(op\ M)$ refers to the unary function constants, and $(op\ M\ N)$ refers to the binary function constants.

Showing that $\leq$ is a pre-congruence is hard. The approach used is similar to Pitts' proof in [Pit94], which uses the technique originally proposed by D. Howe in [How89] . Basically, a similar relation $\leq^*$ is defined that has these pre-congruence properties, and then proved equivalent to $\leq^o$. The relation $\leq^*$ may be defined as follows:

$$
\begin{array}{rcl}
\{\} \vdash \star \leq^*_I N & \iff & \{\} \vdash \star \leq^o_I N \\
\{\} \vdash \underline{n} \leq^*_{\mathbf{nat}} N & \iff & \{\} \vdash \underline{n} \leq^o_{\mathbf{nat}} N \\
\{\} \vdash l \leq^*_{\mathbf{cell}} N & \iff & \{\} \vdash l \leq^o_{\mathbf{cell}} N \\
\Delta \vdash \lambda x\colon\theta.M \leq^*_{\theta \multimap \theta'} N & \iff & \exists M' \ni \Delta, x\colon\theta \vdash M \leq^*_{\theta'} M' \wedge \\
 & & \Delta \vdash \lambda x\colon\theta.M' \leq^o_{\theta \multimap \theta'} N \\
\Delta_1, \Delta_2 \vdash M_1 M_2 \leq^*_{\theta'} N & \iff & \exists M_1', M_2' \ni \Delta_1 \vdash M_1 \leq^*_{\theta \multimap \theta'} M_1' \wedge \\
 & & \Delta_2 \vdash M_2 \leq^*_{\theta} M_2' \wedge \\
 & & \Delta_1, \Delta_2 \vdash M_1' M_2' \leq^o_{\theta'} N \\
\Delta_1, \Delta_2 \vdash M_1 \otimes M_2 \leq^*_{\theta \otimes \theta'} N & \iff & \exists M_1', M_2' \ni \Delta_1 \vdash M_1 \leq^*_{\theta} M_1' \wedge \\
 & & \Delta_2 \vdash M_2 \leq^*_{\theta'} M_2' \wedge \\
 & & \Delta_1, \Delta_2 \vdash M_1' \otimes M_2' \leq^o_{\theta \otimes \theta'} N \\
\Delta_1, \Delta_2 \vdash \mathbf{let}\ M_1\ \mathbf{be}\ x \otimes y\ \mathbf{in}\ M_2 \leq^*_{\theta_3} N & \iff & \exists M_1', M_2' \ni \Delta_1 \vdash M_1 \leq^*_{\theta_1 \otimes \theta_2} M_1' \wedge \\
 & & \Delta_2, x\colon\theta_1, y\colon\theta_2 \vdash M_2 \leq^*_{\theta_3} M_2' \wedge \\
 & & \Delta_1, \Delta_2 \vdash \mathbf{let}\ M_1'\ \mathbf{be}\ x \otimes y\ \mathbf{in}\ M_2' \leq^o_{\theta_3} N \\
\Delta \vdash \mathbf{let}\ \star\ \mathbf{be}\ t\ \mathbf{in}\ M \leq^*_{\theta} N & \iff & \exists M' \ni \Delta \vdash M \leq^*_{\theta} M_1' \wedge \\
 & & \Delta \vdash \mathbf{let}\ \star\ \mathbf{be}\ t\ \mathbf{in}\ M' \leq^o_{\theta} N \\
\Delta_1, \Delta_2 \vdash \langle M_1, M_2 \rangle \leq^*_{\theta \times \theta'} N & \iff & \exists M_1', M_2' \ni \Delta_1 \vdash M_1 \leq^*_{\theta} M_1' \wedge \\
 & & \Delta_2 \vdash M_2 \leq^*_{\theta'} M_2' \wedge \\
 & & \Delta_1, \Delta_2 \vdash \langle M_1', M_2' \rangle \leq^o_{\theta \times \theta'} N \\
\Delta \vdash \pi_1 M \leq^*_{\theta} N & \iff & \exists M' \ni \Delta \vdash M \leq^*_{\theta \times \theta'} M' \wedge \\
 & & \Delta \vdash \pi_1 M' \leq^o_{\theta} N \\
\Delta \vdash \pi_2 M \leq^*_{\theta'} N & \iff & \exists M' \ni \Delta \vdash M \leq^*_{\theta \times \theta'} M' \wedge \\
 & & \Delta \vdash \pi_2 M' \leq^o_{\theta'} N \\
\Delta \vdash \mathit{ifz}\,(\underline{0} \otimes M) \leq^*_{\theta} N & \iff & \exists M' \ni \Delta \vdash M \leq^*_{\theta \times \theta} M' \wedge \\
 & & \Delta \vdash \mathit{ifz}\,(\underline{0} \otimes M') \leq^o_{\theta} N \\
\Delta \vdash \mathit{ifz}\,(\underline{n} \otimes M) \leq^*_{\theta} N & \iff & \exists M' \ni \Delta \vdash M \leq^*_{\theta \times \theta} M' \wedge \\
 & & \Delta \vdash \mathit{ifz}\,(\underline{n} \otimes M') \leq^o_{\theta} N \\
\Delta_1, \Delta_2 \vdash \mathbf{repeat} M_1 M_2 \leq^*_{\mathbf{cell}} N & \iff & \exists M_1', M_2' \ni \Delta_1 \vdash M_1 \leq^*_{\mathbf{cell} \multimap \mathbf{cell} \otimes \mathbf{nat}} M_1' \wedge \\
 & & \Delta_2 \vdash M_2 \leq^*_{\mathbf{cell}} M_2' \wedge \\
 & & \Delta_1, \Delta_2 \vdash \mathbf{repeat} M_1' M_2' \leq^o_{\mathbf{cell}} N \\
\Delta_1, \Delta_2 \vdash \mathit{update}(M_1, \mathit{alloc}(M_2)) \leq^*_{\mathbf{cell}} N & \iff & \exists M_1', M_2' \ni \Delta_1 \vdash M_1 \leq^*_{\mathbf{nat}} M_1' \wedge \\
 & & \Delta_2 \vdash M_2 \leq^*_{\mathbf{nat}} M_2' \wedge \\
 & & \Delta_1, \Delta_2 \vdash \mathit{update}(M_1', \mathit{alloc}(M_2')) \leq^o_{\mathbf{cell}} N \\
\Delta \vdash \mathit{lookup}(\mathit{alloc}(M)) \leq^*_{\mathbf{cell} \otimes \mathbf{nat}} N & \iff & \exists M' \ni \Delta \vdash M \leq^*_{\mathbf{nat}} M' \wedge \\
 & & \Delta \vdash \mathit{lookup}(\mathit{alloc}(M')) \leq^o_{\mathbf{cell} \otimes \mathbf{nat}} N \\
\Delta \vdash \mathit{dealloc}(\mathit{alloc}(M)) \leq^*_I N & \iff & \exists M' \ni \Delta \vdash M \leq^*_{\mathbf{nat}} M' \wedge \\
 & & \Delta \vdash \mathit{dealloc}(\mathit{alloc}(M')) \leq^o_I N
\end{array}
$$

Proposition 16 *If* $\Delta \vdash M \leq^*_\theta N$ *and* $N \leq^o_\theta P$ *then* $\Delta \vdash M \leq^*_\theta P$.

Proof: The proof is by a simple induction over the structure of M for $\Delta \vdash M \leq^*_\theta N$. For each case, the transitivity of $\leq^o$ causes the proof to work. For example, suppose that $\Gamma, \Delta \vdash update(M_1, alloc(M_2)) \leq^*_{\mathbf{cell}} N$. So, by definition of $\leq^*$, $\exists M_1', M_2'$ such that $\Gamma \vdash M_1 \leq^*_{\mathbf{nat}} M_1'$, $\Delta \vdash M_2 \leq^*_{\mathbf{nat}} M_2'$, and $\Gamma, \Delta \vdash update(M_1', alloc(M_2')) \leq^o_{\mathbf{cell}} N$. By the assumption $N \leq^o_\theta P$, and by transitivity of $\leq^o_{\mathbf{cell}}$, $\Gamma, \Delta \vdash update(M_1, alloc(M_2))$ $\leq^*_{\mathbf{cell}} P$. ■

So, by combining Proposition 16 together with the reflexivity of $\leq^*_\theta$ we get the following result:

Proposition 17 *If* $\Delta \vdash M \leq^o_\theta N$, *then* $\Delta \vdash M \leq^*_\theta N$.

And, by a simple induction over the possible structure of P, we get that:

Proposition 18 *If* $\Gamma \vdash M \leq^*_{\theta_1} M'$ *and* $\Delta, x{:}\,\theta_1 \vdash P \leq^*_{\theta_2} P'$, *then* $\Gamma, \Delta \vdash P[M/x] \leq^*_{\theta_2} P'[M'/x]$.

One other important proposition is:

Proposition 19 *If* $- \vdash M \leq^*_\theta N$ *and* $M \rightarrow^*_F v$, *then* $\exists v'$ *such that* $N \rightarrow^*_F v'$ *and* $- \vdash v \leq^*_\theta v'$.

Proof: The proof is by induction over the length of the derivation. The base case, with length of derivation 0, is trivial. For the inductive case, a case analysis is done for which was the n'th rule to fire. The cases are fairly straightforward, if a bit messy. Here are a few of the cases, done in detail:

Case: The reduction $update(M_1, alloc(M_2)) \rightarrow_F alloc(M_1)$. By definition of $\leq^*_\theta$, we have, for $update(M_1, alloc(M_2)) \leq^*_\theta N$:
$\exists M_1', M_2'$, with $M_1 \leq^*_{\mathbf{nat}} M_1'$, $M_2 \leq^*_{\mathbf{nat}} M_2'$, and $update(M_1', alloc(M_2')) \leq^*_\theta N$. Since $update(M_1, alloc(M_2)) \rightarrow^*_F c$, $alloc(M_1) \rightarrow^*_F c$. Since $update(M_1', alloc(M_2')) \rightarrow_F alloc(M_1')$, and since $M_1 \leq^*_{\mathbf{nat}} M_1'$, $\exists c' \ni alloc(M_1') \rightarrow^*_F c'$ with $c \leq^* c'$. And, since

$N \rightarrow_F^* c''$ with $c' \leq^o c''$, since the length of the derivation is shorter, by transitivity and Proposition 17 , we have $c \leq^* c''$.

Case: The reduction **let** $x \otimes y$ **be** $M \otimes N$ **in** $C \rightarrow_F C[M/x, N/y]$. By definition of $\leq_\theta^*$, we have, for **let** $x \otimes y$ **be** M_1 **in** $M_2 \leq_\theta^* N$. $\exists M_1', M_2'$, with $M_1 \leq_{\theta_1 \otimes \theta_2}^* M_1'$, $\{x{:}\,\theta_1, y{:}\,\theta_2\} M_2 \leq_{\mathbf{nat}}^* M_2'$, and **let** $x \otimes y$ **be** M_1' **in** $M_2' \leq_{\theta_3}^o N$. Since **let** $x \otimes y$ **be** M_1 **in** $M_2 \rightarrow_F^* c$, $M_1 \rightarrow_F^* M_{11} \otimes M_{12}$ and $M_2[M_{11}/x, M_{12}/y] \rightarrow_F^* c$. Say that **let** $x \otimes y$ **be** M_1' **in** $M_2' \rightarrow_F^* c'$, and $N \rightarrow_F^* c''$ with $c' \leq^o c''$. Also, suppose that $M_1' \rightarrow_F^* M_{11}' \otimes M_{12}'$. Since $M_1 \leq^* M_1'$, we have that $M_{11} \leq^* M_{11}'$, and $M_{12} \leq^* M_{12}'$. And, by applying Proposition 18 twice, we get that $M_2[M_{11}/x, M_{12}/y] \leq^o M_2'[M_{11}'/x, M_{12}'/y]$, and by transitivity, we are done.

■

By working another induction on the structure of v, we get that:

Proposition 20 *If* $\{\} \vdash v \leq_\theta^* v'$, *then* $v[\leq^*]v'$.

We can finally prove that:

Proposition 21 *If* $\Delta \vdash M \leq_\theta^* N$, *then* $\Delta \vdash M \leq_\theta^o N$.

Proof: We can create the set

$$\mathcal{S} \equiv \{(M, N) \mid \{\} \vdash M \leq_\theta^* N\}$$

By Propositions 19 and 20, $\mathcal{S} \subseteq [\mathcal{S}]$. Thus, for any closed terms, $M \leq_\theta^* N \Longrightarrow M \leq_\theta^o N$. For open terms, we can close them up, as for Proposition 18, and above for the closed terms which result. ■

By inspecting the rules, (and noting that the *Subst* rules follow from Proposition 18) , we get that:

Proposition 22 $\leq^*$ *is a pre-congruence.*

And, finally, by Propositions 17 , 21 , and 22 , we get that:

Proposition 23 $\leq^o$ *is a pre-congruence.*

Using Lemma 15 as the basis for the necessary structural induction, it is possible to prove the following corollary:

Corollary 24 $\Gamma \vdash M \leq M' \wedge \Gamma \vdash N[\cdot]$ *implies that* $\Gamma \vdash N[M] \leq N[M']$.

To complete the proof of the $\Longrightarrow$ direction, recall the definition of observational approximation from Definition 8. Since $\leq = [\leq]$, if $P \leq P'$ and $P \rightarrow^*_F v$, then $P' \rightarrow^*_F v'$ with $v \leq v'$. So, if $M \leq M'$
and $P[M] \rightarrow^*_F v$ then $P[M'] \rightarrow^*_F v'$ with $v \leq v'$. So, $M \sqsubseteq M'$.
■

Proof: (of the $\Longleftarrow$ direction). Here, it is given that $\Gamma \vdash M \sqsubseteq N$. It is necessary to show that $\Gamma \vdash M \leq N$.

It is first necessary to show that $\sqsubseteq$ has similar congruence properties as $\leq$.

Lemma 25

$$\Gamma \vdash M \sqsubseteq M' \wedge \Gamma \vdash N[\cdot] \Rightarrow \Gamma \vdash N[M] \sqsubseteq N[M']$$

Lemma 26 (Congruence of $\sqsubseteq$) $\sqsubseteq$ *has the following congruence properties:*

$$\begin{array}{lll}
\Gamma \vdash M \sqsubseteq M' & \Rightarrow & \Gamma \vdash (op\ M) \sqsubseteq (op\ M') \\
\Gamma \vdash M \sqsubseteq M' \wedge \Gamma \vdash N \sqsubseteq N' & \Rightarrow & \Gamma \vdash (op\ M\ N) \sqsubseteq (op\ M'\ N') \\
\Gamma \vdash M \sqsubseteq M' \wedge \Gamma \vdash N \sqsubseteq N' & \Rightarrow & \Gamma \vdash \langle M, N \rangle \sqsubseteq \langle M', N' \rangle \\
\Gamma \vdash M \sqsubseteq M' \wedge \Gamma \vdash N \sqsubseteq N' & \Rightarrow & \Gamma \vdash M \otimes N \sqsubseteq M' \otimes N' \\
\Gamma \vdash M \sqsubseteq M' \wedge \Gamma \vdash N \sqsubseteq N' & \Rightarrow & \Gamma \vdash \Pi_i \langle M, N \rangle \sqsubseteq \Pi_i \langle M', N' \rangle \\
\Gamma \vdash M \sqsubseteq M' \wedge \Gamma \vdash N \sqsubseteq N' \wedge \Gamma, x, y \vdash T \sqsubseteq T' & \Rightarrow & \Gamma \vdash \mathbf{let}\ M \otimes N\ \mathbf{be}\ x \otimes y\ \mathbf{in}\ T \sqsubseteq \\
 & & \quad \mathbf{let}\ M' \otimes N'\ \mathbf{be}\ x \otimes y\ \mathbf{in}\ T' \\
\Gamma, x \vdash M \sqsubseteq M' & \Rightarrow & \Gamma \vdash \lambda x.M \sqsubseteq \lambda x.M' \\
\Gamma \vdash M \sqsubseteq M' \wedge \Gamma \vdash N \sqsubseteq N' & \Rightarrow & \Gamma \vdash MN \sqsubseteq M'N' \\
\Gamma \vdash M \sqsubseteq M' \wedge \Gamma, x \vdash N \sqsubseteq N' & \Rightarrow & \Gamma \vdash N[M/x] \sqsubseteq N'[M'/x]
\end{array}$$

Proof: The proof for each of the cases is by choosing the appropriate expression, and applying the previous lemma. ■

Lemma 26 allows the following lemma to be proved.

Lemma 27 *The following properties of* $\sqsubseteq$ *hold:*

$$\begin{array}{lll} \lambda x.M \sqsubseteq \lambda x'.M' & \Rightarrow & \forall P.\ \ M[P/x] \sqsubseteq M'[P/x'] \\ \langle P, Q\rangle \sqsubseteq \langle P', Q'\rangle & \Rightarrow & P \sqsubseteq P' \wedge Q \sqsubseteq Q' \\ P \otimes Q \sqsubseteq P' \otimes Q' & \Rightarrow & P \otimes Q \sqsubseteq \bot \vee (P \sqsubseteq P' \wedge Q \sqsubseteq Q') \end{array}$$

This lemma shows that $\sqsubseteq$ is an applicative simulation. So, $Q \sqsubseteq Q' \Rightarrow Q \leq Q'$ for all closed terms. And, if $\{x_1 \colon \theta_1, \ldots, x_n \colon \theta_n\} \vdash M \sqsubseteq M'$, it is possible to repeatedly use the last clause in Lemma 26 to result in $M[\overrightarrow{P}/\overrightarrow{x}] \sqsubseteq M'[\overrightarrow{P}/\overrightarrow{x}]$, so $M[\overrightarrow{P}/\overrightarrow{x}] \leq M'[\overrightarrow{P}/\overrightarrow{x}]$.

■

Thus, operational extensionality is proved.

3.3.2 Summary

In this chapter, we have shown important properties concerning the functional nature of the language of Rudimentary Types. In particular, we have shown subject reduction, adequacy with respect to a stateless denotational model, and an operational extensionality theorem. The next chapter will involve the development of an imperative interpretation of the language, and will show how the imperative and functional interpretations are adequate with respect to one another. This adequacy result will illustrate the simultaneous imperative and functional nature of the language of Rudimentary Types.

Chapter 4

Imperative Semantics and Properties

In this chapter, we develop an imperative operational semantics for the language of Rudimentary Types. We will prove a subject reduction theorem for the imperative operational semantics. We will then prove a coincidence result, which is an adequacy result connecting the imperative and functional operational semantics. Additionally, we show a "no garbage" property of the imperative operational semantics, namely that evaluation of a program always deallocates all allocated cells before it finishes. (Note that this property only refers to the allocation of cells using *alloc*, and does *not* say that a garbage collector would not be needed anywhere in an implementation.)

4.1 Operational Semantics

In Chapter 3, we developed a functional operational semantics. Here, we use the same reductions of Figure 2 to create an imperative operational semantics. The major difference between the functional and imperative descriptions occurs with the interpretation of the **cell** functions (*alloc*, *dealloc*, *update*, and *lookup*).

The imperative semantics uses an explicit state parameter, where a state is a finite partial function $s: L \to N$ from a finite set of locations to the natural numbers.

Additionally, we enrich the set of pre-terms so that all $\ell \in L$ are added as additional constants of type **cell**. By doing this, we are actually defining a new language.

Definition 28 *We write $\Delta \vdash^{\mathrm{imp}} M{:}\,\theta$ to indicate that $\Delta \vdash M{:}\,\theta$ is derivable using the rules and constants of Chapter 2 , together with constants $\ell{:}\,\mathbf{cell}$ for each $\ell \in L$.*

Reductions in the imperative operational semantics take the form

$$M, s \;\triangleright\; M', s'$$

where M, M' are terms, and s, s' are states. The semantics is specified in Figure 5. The change of state in the case of *dealloc* is simply the removal of the location ℓ from the set of locations accessible to the program. The requirement that $\ell \in \mathit{dom}\, s$ in rules for *lookup*, *update* and *dealloc* means that programs trying to use locations not in the store get stuck. For instance, the configuration $\mathit{dealloc}(\ell), \{\}$ cannot be reduced further, where $\{\}$ is the empty state.

$$\frac{M \;\rightarrow\; N}{M, s \;\triangleright\; N, s}$$

$$\frac{M, s \;\triangleright\; M', s'}{C[M], s \;\triangleright\; C[M'], s'}$$

$$\begin{array}{llll}
\mathit{update}(\underline{n}, \ell),\ s & \triangleright & \ell,\ (s \mid \ell \mapsto n) & (\ell \in \mathit{dom}\, s) \\
\mathit{lookup}(\ell),\ s & \triangleright & (\ell, s(\ell)),\ s & (\ell \in \mathit{dom}\, s) \\
\mathit{dealloc}(\ell),\ s & \triangleright & \star,\ s - \ell & (\ell \in \mathit{dom}\, s) \\
\mathit{alloc}(\underline{n}),\ s & \triangleright & \ell, (s \mid \ell \mapsto n) & (\ell \notin \mathit{dom}\, s)
\end{array}$$

Figure 5: Imperative Operational Semantics

We also need to add the **repeat** reductions to our list of possible reductions. In the case that applying f to l results in a tensor pair, $\underline{n} \otimes l'$, if $n \neq 0$, we need to go through the **repeat** loop again. (Note that in the application of the function f to the argument l, it is possible for the state to change.) As such, we write the rule as the following:

$$\frac{(f\ l), s \ \triangleright^k\ \underline{n} \otimes l', s'}{\mathbf{repeat}\ f\ l, s\ \triangleright^{(k+1)}\ \mathbf{repeat}\ f\ l', s'}\ n \neq 0$$

Note that, as in Chapter 3, $\triangleright^k$ means "reduces in k steps", whereas $\triangleright$ means "reduces in 1 step". Additionally, $\triangleright^*$ is used to mean "reduces in 0 or more steps". The other case for **repeat**, when applying f to l results in the tensor pair $\underline{0} \otimes l'$ can be written as:

$$\frac{(f\ l), s \ \triangleright^k\ \underline{0} \otimes l', s'}{\mathbf{repeat}\ f\ l, s\ \triangleright^{(k+1)}\ l', s'}$$

Let us consider again the example from the Introduction, and consider its behavior in the imperative semantics.

$$\begin{array}{l} (\lambda x\,.\ \mathbf{let}\ a \otimes m\ \mathbf{be}\ lookup(update\,\underline{7}\,x)\ \mathbf{in} \\ \qquad\quad \mathbf{let}\ b \otimes n\ \mathbf{be}\ lookup(update\,\underline{8}\,x)\ \mathbf{in}\ m+n \\ )\ \ (alloc\,\underline{0}) \end{array}$$

In the imperative semantics there are three possible final answers for this term: 14, 15 and 16. For example, here is a reduction sequence which yields 16. The first step in the initial reduction could be the evaluation of the $(alloc\,\underline{0})$, resulting in

$$\begin{array}{l} (\lambda x\,.\ \mathbf{let}\ a \otimes m\ \mathbf{be}\ lookup(update\,\underline{7}\,x)\ \mathbf{in} \\ \qquad\quad \mathbf{let}\ b \otimes n\ \mathbf{be}\ lookup(update\,\underline{8}\,x)\ \mathbf{in}\ m+n \\ )\ \ \ell, (s \mid \ell \mapsto \underline{0}). \end{array}$$

The next step could be the function application resulting in

$$\begin{array}{l} \mathbf{let}\ a \otimes m\ \mathbf{be}\ lookup(update\,\underline{7}\,\ell)\ \mathbf{in} \\ \quad \mathbf{let}\ b \otimes n\ \mathbf{be}\ lookup(update\,\underline{8}\,\ell)\ \mathbf{in}\ m+n, (s \mid \ell \mapsto \underline{0}). \end{array}$$

Evaluating the first *update* results in

$$\begin{array}{l}\textbf{let } a \otimes m \textbf{ be } \mathit{lookup}(\ell) \textbf{ in} \\ \quad \textbf{let } b \otimes n \textbf{ be } \mathit{lookup}(\mathit{update}\,\underline{8}\,\ell) \textbf{ in } m + n, (s \mid \ell \mapsto \underline{7}).\end{array}$$

Then, evaluating the second *update* results in

$$\begin{array}{l}\textbf{let } a \otimes m \textbf{ be } \mathit{lookup}(\ell) \textbf{ in} \\ \quad \textbf{let } b \otimes n \textbf{ be } \mathit{lookup}(\ell) \textbf{ in } m + n, (s \mid \ell \mapsto \underline{8}).\end{array}$$

Then, evaluating the first *lookup* results in

$$\begin{array}{l}\textbf{let } a \otimes m \textbf{ be } \ell \otimes \underline{8} \textbf{ in} \\ \quad \textbf{let } b \otimes n \textbf{ be } \mathit{lookup}(\ell) \textbf{ in } m + n, (s \mid \ell \mapsto \underline{8}).\end{array}$$

Next, an evaluation of the second *lookup* occurs, resulting in

$$\begin{array}{l}\textbf{let } a \otimes m \textbf{ be } \ell \otimes \underline{8} \textbf{ in} \\ \quad \textbf{let } b \otimes n \textbf{ be } \ell \otimes \underline{8} \textbf{ in } m + n, (s \mid \ell \mapsto \underline{8}).\end{array}$$

Evaluating the second **let** results in

$$\begin{array}{l}\textbf{let } a \otimes m \textbf{ be } \ell \otimes \underline{8} \textbf{ in} \\ \quad (m + n)[\underline{8}/n], (s \mid \ell \mapsto \underline{8}).\end{array}$$

Finally, evaluating the first **let** results in the expression

$$(m + n)[\underline{8}/n][\underline{8}/m], (s \mid \ell \mapsto \underline{8}),$$

which evaluates to 16. The functional meaning of this term is 16, so the imperative and functional interpretations of this term differ.

Also, at the end of any evaluation there is "garbage", in that the allocated cell has not been deallocated. For untyped pre-terms (such as the above example), the imperative semantics may be indeterminate. As a result of this, the imperative operational semantics cannot match the functional operational semantics in general.

However, this example does not type check. There are two occurrences of x in the body of the function (Contraction is not allowed), and the two cells, a and b are not explicitly deallocated (Weakening is not allowed). So, this pre-term (for which the functional and imperative interpretations may give different results) will not be typable in our language.

4.2 Coincidence

In this section, we show that, for *well-typed terms*, there is a close relationship between the two operational semantics. We formalize a coincidence theorem, which says that well-typed terms have the same behavior in both semantics.

Theorem 29 (Coincidence) *If* $- \vdash^{\mathrm{imp}} M : \mathbf{nat}$*, then*

$$M \to_F^* \underline{n} \iff \exists s\,(M, \{\} \rhd^* \underline{n}, s).$$

A consequence of coincidence is that, for well-typed terms in the imperative semantics, there is no indeterminacy, despite the different order of reductions. The remainder of this section is devoted to proving coincidence. To prove coincidence we must first prove a subject reduction theorem for the imperative operational semantics, as well as a few other related lemmas. In the course of proving coincidence, we also prove the following theorem:

Theorem 30 (Progress) *If* $- \vdash^{\mathrm{imp}} M : \mathbf{nat}$ *and* $M, \{\} \rhd^+ N, s$ *then* $\exists M', s'$ *such that* $M, s \rhd M', s'$.

The progress theorem states that well-typed programs never get stuck.

4.2.1 Preliminaries

We define

$$\Delta \vdash_L^{\mathrm{imp}} M : \theta \;\equiv_{def}\; \Delta \vdash^{\mathrm{imp}} M : \theta \wedge freeloc(M) \subseteq L,$$

where $freeloc(M)$ refers to the location constants that appear in M. In other words, $\Delta \vdash_L^{\mathrm{imp}} M : \theta$ if M is typable and involves only locations in L.

To prove subject reduction for the imperative operational semantics, a few propositions are needed. The proofs are omitted as they are mostly similar to those presented by Barber in [Bar96] .

Definition 31 (Uniqueness for Locations) *Consider the judgement* $\Delta \vdash_L^{\mathrm{imp}} t : \theta$. *We say that* $\ell \in L$ *occurs uniquely in* t *if and only if* ℓ *appears exactly once in* t*, or* *else the only appearances of ℓ in t are in a subterm, $\langle M, N\rangle$, and ℓ occurs uniquely in both M and N.*

The following proposition states that locations (which are how variables of type **cell** get handled in the imperative operational semantics) have a similar uniqueness property to the variables as described in Section 2.2. The difference is that while variables *must* occur uniquely within a term, locations need not occur within a term.

Proposition 32 (Free Locations) *If $\Delta \vdash_L^{\mathrm{imp}} t\colon \theta'$, and $\ell \in L$, then ℓ occurs uniquely in t or it does not appear in t at all.*

Proposition 33 (Location Weakening) *If $\Delta \vdash_{L_1}^{\mathrm{imp}} t\colon \theta$, then $\Delta \vdash_{L_1 \cup L_2}^{\mathrm{imp}} t\colon \theta$.*

4.2.2 Subject Reduction

Before showing coincidence, we need to prove subject reduction for the imperative operational semantics. Subject reduction for the functional operational semantics was proved in Chapter 3.

Theorem 34 (Subject Reduction for imperative operational semantics) *If $\Delta \vdash_L^{\mathrm{imp}} t\colon \theta$ and $t, s \triangleright^* u, s'$, then there exists an L' such that $\Delta \vdash_{L'}^{\mathrm{imp}} u\colon \theta$.*

Again, it is noted that the expression $t, s \triangleright^* u, s'$ means that the term t in state s reduces to the term u in state s' in zero or more steps.

Proof: Like Proposition 5 from Section 3.1.2, the proof is an inductive one over on the structure of the derivation of the judgement of $t, s \triangleright^k u, s'$. The proof is accomplished by considering each of the possible reduction rules that could have been applied to the term t.

The cases when t is $(\lambda x.M)N$, **let** $x \otimes y$ **be** $M \otimes N$ **in** C, and $\pi_i\langle M_1, M_2\rangle$ are uninteresting, especially since no change of state is involved. Additionally, the cases when t is of the form *ifz* $(\underline{n} \otimes M)$, $+\,(\underline{n} \otimes \underline{m})$, and $\dot{-}\ (\underline{n} \otimes \underline{m})$ are straightforward as no change of state is involved.

Case: t is reduced by $update(\underline{n}, \ell), s \triangleright \ell,\ (s \mid \ell \mapsto n)$.
The term t must have had a derivation as follows:

$$\frac{\Delta \vdash^{\text{imp}}_{L} \ell \otimes \underline{n} \colon \mathbf{cell} \otimes \mathbf{nat} \quad \{\} \vdash^{\text{imp}}_{\{\}} update \colon \mathbf{cell} \otimes \mathbf{nat} \multimap \mathbf{cell}}{\Delta \vdash^{\text{imp}}_{L} update(\ell, \underline{n}) \colon \mathbf{cell}}$$

The second premise in this derivation is simply the result of *update* being a constant. The first premise of this derivation must have had a derivation as follows:

$$\frac{\Delta \vdash^{\text{imp}}_{L} \ell \colon \mathbf{cell} \quad \{\} \vdash^{\text{imp}}_{\{\}} \underline{n} \colon \mathbf{nat}}{\Delta \vdash^{\text{imp}}_{L} \ell \otimes \underline{n} \colon \mathbf{cell} \otimes \mathbf{nat}}$$

Thus, $\Delta \vdash^{\text{imp}} \ell \colon \mathbf{cell}$ as necessary.

Case: t is reduced by $lookup(\ell), s \triangleright (\ell, s(\ell)), s$.
It must be that t had a derivation as follows:

$$\frac{\Delta \vdash^{\text{imp}} \ell \colon \mathbf{cell} \quad \{\} \vdash^{\text{imp}} lookup \colon \mathbf{cell} \multimap \mathbf{cell} \otimes \mathbf{nat}}{\Delta \vdash^{\text{imp}} lookup(\ell) \colon \mathbf{cell} \otimes \mathbf{nat}}$$

Since $s(\ell)$ is the value of the **nat** at location ℓ,

$$\Delta \vdash^{\text{imp}} \ell \otimes s(\ell) \colon \mathbf{cell} \otimes \mathbf{nat}.$$

Case: t is reduced by $dealloc(\ell), s \triangleright \star, s - \ell$.
Since *dealloc* and ℓ are both constants, the only Δ for which $\Delta \vdash^{\text{imp}} dealloc(\ell) \colon \star$ is $\Delta = \{\}$ (because of uniqueness). Since $\star$ is a constant,

$$- \vdash^{\text{imp}} \star \colon I.$$

Case: t is reduced by $alloc(\underline{n}), s \triangleright \ell, (s \mid \ell \mapsto n)$.
Note that $\Delta \vdash^{\text{imp}}_{L} alloc(\underline{n}) \colon \mathbf{cell}$ implies that $\Delta \vdash^{\text{imp}}_{L \cup \{\ell\}} \ell \colon \mathbf{cell}$.

Case: t is reduced by

$$\frac{(f\ \ell), s \ \triangleright^{k}\ \underline{n} \otimes \ell', s'}{\mathbf{repeat}\ f\ \ell, s\ \triangleright^{(k+1)}\ \mathbf{repeat}\ f\ \ell', s'}\ n <> 0$$

Here, it is necessary to show that $\Delta \vdash_{L'}^{\text{imp}} \mathbf{repeat}\ f\ \ell' \colon \mathbf{cell}$ if $\Delta \vdash_{L}^{\text{imp}} \mathbf{repeat}\ f\ \ell \colon \mathbf{cell}$. The term $\mathbf{repeat}\ f\ \ell \colon \mathbf{cell}$ must have been the result of the following function application:

$$\frac{\Delta \vdash_{L}^{\text{imp}} \mathbf{repeat}\ f \colon \mathbf{cell} \multimap \mathbf{cell} \quad \Delta' \vdash_{L'}^{\text{imp}} \ell \colon \mathbf{cell}}{\Delta, \Delta' \vdash_{L \cup L'}^{\text{imp}} \mathbf{repeat}\ f\ \ell \colon \mathbf{cell}}$$

Due to uniqueness (Proposition 3 from Chapter 2), $\Delta' = \{\}$. The function application may be rewritten as:

$$\frac{\Delta \vdash_{L}^{\text{imp}} \mathbf{repeat}\ f \colon \mathbf{cell} \multimap \mathbf{cell} \quad - \vdash_{L'}^{\text{imp}} \ell \colon \mathbf{cell}}{\Delta \vdash_{L \cup L'}^{\text{imp}} \mathbf{repeat}\ f\ \ell \colon \mathbf{cell}}$$

But, since $- \vdash_{L''}^{\text{imp}} \ell' \colon \mathbf{cell}$, it is possible to immediately apply the following function application to obtain the desired result:

$$\frac{\Delta \vdash_{L}^{\text{imp}} \mathbf{repeat}\ f \colon \mathbf{cell} \multimap \mathbf{cell} \quad - \vdash_{L''}^{\text{imp}} \ell' \colon \mathbf{cell}}{\Delta \vdash_{L \cup L''}^{\text{imp}} \mathbf{repeat}\ f\ \ell' \colon \mathbf{cell}}$$

The other **repeat** case is slightly different, and takes advantage of the induction hypothesis.

Case: t is reduced by

$$\frac{(f\ \ell), s \ \triangleright^{k}\ \underline{0} \otimes \ell', s'}{\mathbf{repeat}\ f\ \ell, s \ \triangleright^{(k+1)}\ \ell', s'}$$

Here, it is necessary to show that $\Delta \vdash_{L'}^{\text{imp}} \ell' \colon \mathbf{cell}$ if $\Delta \vdash_{L}^{\text{imp}} \mathbf{repeat}\ f\ \ell \colon \mathbf{cell}$.

Since the structure of the judgement of $f\ \ell, s \triangleright^{k} \underline{0} \otimes \ell', s'$ is "simpler", by the induction hypothesis we have that $\Delta \vdash_{L''}^{\text{imp}} \underline{0} \otimes \ell' \colon \mathbf{nat} \otimes \mathbf{cell}$ when $\Delta \vdash_{L}^{\text{imp}} f\ \ell \colon \mathbf{nat} \otimes \mathbf{cell}$. By uniqueness, we know that $\Delta = \{\}$. Now the term $\underline{0} \otimes \ell'$ may have been constructed as follows:

$$\frac{- \vdash_{\{\}}^{\text{imp}} \underline{0} \colon \mathbf{nat} \quad - \vdash_{L''}^{\text{imp}} \ell' \colon \mathbf{cell}}{- \vdash_{L''}^{\text{imp}} \underline{0} \otimes \ell' \colon \mathbf{nat} \otimes \mathbf{cell}}$$

Because of uniqueness, we have that: $- \vdash_{L''}^{\text{imp}} \ell' \colon \mathbf{cell}$.

Case: t is reduced by

$$\frac{M, s \triangleright M', s'}{C[M], s \triangleright C[M'], s'}$$

This case just amounts to an induction based on contexts (and within each context a structural induction is done over the possible reductions). Thus, this case amounts to a redoing of all of the previous cases, but no added difficulty is involved. ■

4.2.3 Proof of Coincidence

For ease of proving coincidence between the imperative and functional operational semantics, it is useful — in the functional operational semantics — to "tag" each occurrence of an *alloc* with a label l, and adding a new constant, $alloc_l$, for each $l \in L$. The language is extended with a set of new constants, $\{alloc_l \mid l \in L\}$. As this tagging produces a new language, we need two new definitions.

Definition 35 *We write $\Delta \vdash^f M\colon\theta$ to indicate that $\Delta \vdash M\colon\theta$ is derivable using the rules and constants of Chapter 2 , together with constants $alloc_l$.*

Definition 36 *We write $M \rightarrow_{F_l} N$ to indicate that $M \rightarrow^*_{F_l} N$ is derivable using the reduction rules from Figure 6.*

We note that "where l is a new label" is not mathematically rigorous. To be formal, we would need to discuss terms of the form M, L, where L is the set of labels associated with *alloc*s in M. Then, reductions would look like $M, L \rightarrow_{F_l} M', L'$, and the *alloc* reduction would be properly stated as: $alloc(\underline{n}), L \rightarrow_{F_l} alloc_l(\underline{n}), L \cup l$ *where* $l \notin L$. The proof of Lemma 42 assumes this more formal definition.

Proposition 37 *For all typable terms M, $M \rightarrow^*_F \underline{n}$ in the functional operational semantics described without labels if and only if $M \rightarrow^*_{F_l} \underline{n}$ in the functional operational semantics with labels, as described in Figure 6.*

Proof: [Sketch] The proof is by a straightforward induction. In the $\Rightarrow$ direction, when an *alloc* is to be reduced, the additional *alloc* rule in Figure 6 is applied. Otherwise the reductions are identical. In the $\Leftarrow$ direction, it is sufficient to simply remove the labels, and remove the steps involving the last rule in Figure 6. ■

$$
\begin{array}{rll}
(\lambda x.M)N & \to_{F_l} & M[N/x] \\[1em]
\mathbf{let}\ x \otimes y\ \mathbf{be}\ M \otimes N\ \mathbf{in}\ C & \to_{F_l} & C[M/x, N/y] \\[1em]
\pi_i\langle M_1, M_2\rangle & \to_{F_l} & M_i \\[1em]
update(\underline{n},\ alloc_l(\underline{m})) & \to_{F_l} & alloc_l(\underline{n}) \\[1em]
lookup(alloc_l(\underline{n})) & \to_{F_l} & \underline{n} \otimes alloc_l(\underline{n}) \\[1em]
dealloc(alloc_l(\underline{n})) & \to_{F_l} & \star \\[1em]
alloc(\underline{n}) & \to_{F_l} & alloc_l(\underline{n})\ where\ l\ is\ a\ new\ label
\end{array}
$$

$$
\frac{M \to_{F_l} N}{C[M] \to_{F_l} C[N]}
$$

$$
\frac{M \to N}{M \to_{F_l} N}
$$

Figure 6: β reduction rules for the Functional Operational Semantics with Labels

We define the notation,

$$\Delta \vdash_L^f M\colon\theta \ \equiv_{def}\ \Delta \vdash^f M\colon\theta \wedge freeloc(M) \subseteq L$$

where $freeloc(M)$ refers to the tags to the *allocs* in the functional language.

Before proving adequacy of the imperative and functional versions of the language, it is first necessary to prove some technical lemmas.

Lemma 38 *For well-typed terms in the imperative operational semantics, every reduction rule except for alloc, dealloc, and* **repeat** *preserves the set of locations. In other words, if* $\Delta \vdash_L^{\mathrm{imp}} t\colon\theta$ *,* $t, s \triangleright u, s'$ *(and the reduction that occurs does NOT involve an alloc, a dealloc, or a* **repeat***), then* $\Delta \vdash_L^{\mathrm{imp}} u\colon\theta$*.*

Lemma 39 *For well-typed terms in the functional operational semantics, every reduction rule except for alloc, dealloc, and* **repeat** *preserves the set of locations. In other words, if* $t\colon\theta$ *is well typed, if* $\Delta \vdash_L^f t\colon\theta$ *,* $t \to_{F_l} u$ *(and the reduction that occurs does NOT involve an alloc, a dealloc, or a* **repeat***), then* $\Delta \vdash_L^f u\colon\theta$*.*

The proof of these lemmas can be immediately seen by looking at the reduction rules in the proof of Subject Reduction. In each case (except for the *alloc*, *dealloc*, and **repeat** reduction rules), the set of locations used in making up the premises is identical to the set of locations used in the result, after the reduction has occurred.

Lemma 40 *For well-typed terms, no reduction rules in the imperative semantics duplicates locations. In other words, if* $\Delta \vdash_L^{\mathrm{imp}} t\colon\theta$*,* $t, s \triangleright^* u, s'$*, and* $\Delta' \vdash_{L'}^{\mathrm{imp}} u\colon\theta'$*, if* $\ell \in L$ *and* ℓ *appears uniquely in* t*, then* ℓ *appears either uniquely, or not at all in* u*. (See Definitions 2 and 31 for the definition of unique.)*

This lemma may seem surprising, in that it might seem possible that locations could be duplicated, such as in the example:

$$(\lambda x.\lambda y.y(xx))l \to \lambda y.y(ll)$$

But, since none of the rules for building terms allow the duplication of locations, there is no way to get the body of the function to duplicate the argument (and thus to potentially duplicate locations). The only allowed duplication of a term is via a copy (in which the object being copied must be of type **nat**), or via pairing (which, because of uniqueness, really means that only one of the pair can ever be selected, does not result in the duplication of locations). In the example above, the only way, in the type system, that x can be duplicated is if its type is **nat**.

Proof: The proof is by a case analysis over the possible reductions. Formally, the proof is nothing more than applying the cut lemma to the term being reduced, and then using Proposition 3 (from Section 2.2) as a means of guaranteeing uniqueness. Because the case of $\multimap$ elimination (when the function is *not* a function constant) is not trivial, it is included in detail.

Case: t is an application. Then t must have been formed by the $\multimap$-E rule:

$$\frac{\Delta_1 \vdash^{\text{imp}}_{L_1} \lambda x{:}\, \theta'.P{:}\, \theta' \multimap \theta \quad \Delta_2 \vdash^{\text{imp}}_{L_2} Q{:}\, \theta'}{\Delta_1, \Delta_2 \vdash^{\text{imp}}_{L_1 \cup L_2} P(Q){:}\, \theta} \multimap - E$$

The $\lambda x.P$ term must have been created by the $\multimap$-I rule:

$$\frac{\Delta_1, x{:}\, \theta' \vdash^{\text{imp}}_{L_1} P{:}\, \theta}{\Delta_1 \vdash^{\text{imp}}_{L_1} \lambda x{:}\, \theta'.P{:}\, \theta' \multimap \theta} \multimap - I$$

By Proposition 4 , it is possible to cut the premise of this rule with the second premise of the previous rule, obtaining:

$$\Delta_1, x{:}\, \theta' \vdash^{\text{imp}}_{L_1} P{:}\, \theta \;\; \textit{and} \;\; \Delta_2 \vdash^{\text{imp}}_{L_2} Q{:}\, \theta' \Longrightarrow \Delta_1, \Delta_2 \vdash^{\text{imp}}_{L_1 \cup L_2} P[Q/x]{:}\, \theta$$

By Proposition 3 , x occurs uniquely, or not at all in P. ■

Lemma 41 *In the functional operational semantics, no labeled allocs (i.e., locations) are duplicated in any well typed term.*

Proof: The proof is very similar to the previous lemma. ■

At this point, we remark that Lemmas 40 and 41 immediately imply the Progress result, Theorem 30.

The next lemma proves a more general property, of which coincidence will follow as a special case.

Lemma 42 *Suppose that M is a term with location constants in which each location occurs uniquely. Suppose that s is a state, and that $freeloc(M) \subseteq dom(s)$. Then,* $\exists s'.M, s \rhd^* \underline{n}, s' \iff M^o \rightarrow^*_{F_l} \underline{n}$*, where* $M^o = M$ *with* $(alloc_l(\underline{n})/\ell)$*.*

Proof: The proof is by induction on the length of the reduction sequence $M^o \rightarrow^*_{F_l} \underline{n}$. For the base case, zero steps, $\underline{n} \rightarrow^*_{F_l} \underline{n}$, and $\underline{n}, s \rhd^* \underline{n}, s$. The inductive case is itself a case analysis, based on which rule fired first. The actual proof itself is a sort of "lock step" argument, where the actual reductions match up one by one between the functional and the imperative semantics.

Case: $C[dealloc(alloc_l(\underline{n}))]^o \rightarrow_{F_l} C[\star]^o$.

Since $C[\star]^o \rightarrow^*_{F_l} \underline{n}$ has a shorter derivation length,

$$C[\star] \rightarrow^*_{F_l} \underline{n} \iff \exists s'.C[\star], s_{C[\star]} \rhd^* \underline{n}, s'$$

There are two possible subcases:

If $C[\star]^o \rightarrow^*_{F_l} \underline{n}$,

$$C[dealloc(alloc_l(\underline{n}))]^o \rightarrow_{F_l} C[\star]^o \rightarrow^*_{F_l} \underline{n}$$

So, using the induction hypothesis,

$$C[\star], s_{C[\star]} \rhd^* \underline{n}, s'$$

combined with

$$C[dealloc(alloc(\underline{n}))], s_{C[dealloc(alloc(\underline{n}))]} \rhd C[\star], s_{C[dealloc(alloc(\underline{n}))]-\ell}$$

the desired result is achieved, since $s_{C[dealloc(alloc(\underline{n}))]-\ell} = s_{C[\star]}$ using the uniqueness of $C[\star]$ labels and Lemmas 40 and 41.

If $C[\star]^o \not\rightarrow^*_{F_l} \underline{n}$, $C[dealloc(alloc_l(\underline{n}))]^o \not\rightarrow^*_{F_l} \underline{n}$. By the induction hypothesis,

$$C[\star], s_{C[\star]} \not\triangleright^* \underline{n}, s',\ any\ s'.$$

And, combining this with the argument above,

$$C[dealloc(alloc_l(\underline{n}))], s_{C[dealloc(alloc_l(\underline{n}))]} \not\triangleright^* \underline{n}, s',\ any\ s'.$$

Case: $C[lookup(alloc_l(\underline{n}))]^o \rightarrow_{F_l} C[alloc_l(\underline{n}) \otimes \underline{n}]^o$.

Since $C[alloc_l(\underline{n}) \otimes \underline{n}]^o \rightarrow^*_{F_l} \underline{n}$ has a shorter derivation length,

$$C[alloc_l(\underline{n}) \otimes \underline{n}]^o \rightarrow^*_{F_l} \underline{n'} \iff \exists s'.C[alloc_l(\underline{n}) \otimes \underline{n}], s_{C[alloc_l(\underline{n})\otimes\underline{n}]} \triangleright^* \underline{n'}, s'$$

There are two possible subcases:

If $C[alloc_l(\underline{n}) \otimes \underline{n}]^o \rightarrow^*_{F_l} \underline{n'}$,

$$C[lookup(alloc_l(\underline{n}))]^o \rightarrow_{F_l} C[alloc_l(\underline{n}) \otimes \underline{n}]^o \rightarrow^*_{F_l} \underline{n'}$$

So, using the induction hypothesis,

$$C[alloc_l(\underline{n}) \otimes \underline{n}], s_{C[alloc_l(\underline{n})\otimes\underline{n}]} \triangleright^* \underline{n'}, s'$$

combined with

$$C[lookup(alloc(\underline{n}))], s_{C[lookup(alloc(\underline{n}))]} \triangleright C[s(alloc(\underline{n})) \otimes \underline{n}], s_{C[lookup(alloc(\underline{n}))]}$$

the desired result is achieved, since $s_{C[lookup(alloc(\underline{n}))]} = s_{C[alloc(\underline{n})\otimes\underline{n}]}$ and $s(alloc(\underline{n})) = alloc(\underline{n})$ as the set of locations is preserved (since the non-*alloc* and non-*dealloc* reduction rules preserve the set of locations by Lemmas 38 and 39).

If $C[alloc_l(\underline{n}) \otimes \underline{n}]^o \not\rightarrow^*_{F_l} \underline{n'}$, $C[lookup(alloc_l(\underline{n}))]^o \not\rightarrow^*_{F_l} \underline{n'}$. By the induction hypothesis,

$$C[alloc_l(\underline{n}) \otimes \underline{n}], s_{C[alloc_l(\underline{n})\otimes\underline{n}]} \not\triangleright^* \underline{n'}, s',\ any\ s'.$$

And, combining this with the argument above,

$$C[lookup(alloc_l(\underline{n}))], s_{C[lookup(alloc_l(\underline{n}))]} \not\triangleright^* \underline{n'}, s',\ any\ s'.$$

Case: $C[update(\underline{n}, alloc_l(\underline{m}))]^o \rightarrow_{F_l} C[alloc_l(\underline{n})]^o$.

Since $C[alloc_l(\underline{n})]^o \rightarrow^*_{F_l} \underline{n}$ has a shorter derivation length,

$$C[alloc_l(\underline{n})]^o \rightarrow^*_{F_l} \underline{n'} \iff \exists s'.C[alloc_l(\underline{n})], s_{C[alloc_l(\underline{n})]} \triangleright^* \underline{n'}, s'$$

There are two possible subcases:

If $C[alloc_l(\underline{n})]^o \rightarrow^*_{F_l} \underline{n'}$,

$$C[update(\underline{n}, alloc_l(\underline{m}))]^o \rightarrow_{F_l} C[alloc_l(\underline{n})]^o \rightarrow^*_{F_l} \underline{n'}$$

So, using the induction hypothesis,

$$C[alloc_l(\underline{n})], s_{C[alloc_l(\underline{n})]} \triangleright^* \underline{n'}, s'$$

combined with

$$\begin{array}{l} C[update(\underline{n}, alloc(\underline{m}))], s_{C[update(\underline{n},alloc(\underline{m}))]} \triangleright \\ \quad C[alloc(\underline{m})], s_{C[update(\underline{n},alloc(\underline{m}))]} \mid alloc(\underline{m}) \mapsto \underline{n} \end{array}$$

the desired result is achieved, since $s_{C[update(\underline{n},alloc(\underline{m}))]} \mid alloc(\underline{m}) \mapsto \underline{n} = s_{C[alloc(\underline{n})]}$ as the set of locations is preserved (since the non-*alloc* and non-*dealloc* reduction rules preserve the set of locations by Lemmas 38 and 39). And, the value in the array $alloc(\underline{m})$ in the state,
$s_{C[update(\underline{n},alloc(\underline{m}))]} \mid alloc(\underline{m}) \mapsto \underline{n}$ is n, as is $alloc(\underline{n})$ in the state, $s_{C[alloc(\underline{n})]}$.

If $C[alloc_l(\underline{n})]^o \not\rightarrow^*_{F_l} \underline{n'}$, $C[update(\underline{n}, alloc_l(\underline{m}))]^o \not\rightarrow^*_{F_l} \underline{n'}$. By the induction hypothesis,

$$C[alloc_l(\underline{n})], s_{C[alloc_l(\underline{n})]} \not\triangleright^* \underline{n'}, s',\ any\ s'.$$

And, combining this with the argument above,

$$C[update(\underline{n}, alloc_l(\underline{m}))], s_{C[update(\underline{n}, alloc_l(\underline{m}))]} \not\triangleright^* \underline{n'}, s',\ any\ s'.$$

Case: $C[(\lambda x.M)N]^o \to_{F_l} C[M[N/x]]^o$.

Since $C[M[N/x]]^o \to^*_{F_l} \underline{n}$ has a shorter derivation length,

$$C[M[N/x]]^o \to^*_{F_l} \underline{n} \iff \exists s'.C[M[N/x]], s_{C[M[N/x]]} \triangleright^* \underline{n}, s'$$

There are two possible subcases:

If $C[M[N/x]]^o \to^*_{F_l} \underline{n}$,

$$C[(\lambda x.M)N]^o \to_{F_l} C[M[N/x]]^o \to^*_{F_l} \underline{n}$$

So, using the induction hypothesis,

$$C[M[N/x]], s_{C[M[N/x]]} \triangleright^* \underline{n}, s'$$

combined with

$$C[(\lambda x.M)N], s_{C[(\lambda x.M)N]} \triangleright C[M[N/x]], s_{C[M[N/x]]}$$

the desired result is achieved, since $s_{C[(\lambda x.M)N]} = s_{C[M[N/x]]}$ since there is no state change during this reduction.

If $C[M[N/x]]^o \not\to^*_{F_l} \underline{n}$, $C[(\lambda x.M)N]^o \not\to^*_{F_l} \underline{n}$. By the induction hypothesis,

$$C[M[N/x]], s_{C[M[N/x]]} \not\triangleright^* \underline{n}, s',\ any\ s'.$$

And, combining this with the argument above,

$$C[(\lambda x.M)N], s_{C[(\lambda x.M)N]} \not\triangleright^* \underline{n}, s',\ any\ s'.$$

Case: $C[\textbf{repeat}\ f\ \ell]^o \rightarrow_{F_l} C[\textbf{repeat}\ f\ \ell']^o$.

Since $C[\textbf{repeat}\ f\ \ell']^o \rightarrow^*_{F_l} \underline{n}$ has a shorter derivation length,

$$C[\textbf{repeat}\ f\ \ell'] \rightarrow^*_{F_l} \underline{n} \iff \exists s'.C[\textbf{repeat}\ f\ \ell'], s_{C[\textbf{repeat}\ f\ \ell']} \triangleright^* \underline{n}, s'$$

There are two possible subcases:

If $C[\textbf{repeat}\ f\ \ell']^o \rightarrow^*_{F_l} \underline{n}$,

$$C[\textbf{repeat}\ f\ \ell]^o \rightarrow_{F_l} C[\textbf{repeat}\ f\ \ell']^o \rightarrow^*_{F_l} \underline{n}$$

So, using the induction hypothesis,

$$C[\textbf{repeat}\ f\ \ell'], s_{C[\textbf{repeat}\ f\ \ell']} \triangleright^* \underline{n}, s'$$

combined with

$$C[\textbf{repeat}\ f\ \ell], s_{C[\textbf{repeat}\ f\ \ell]} \triangleright C[\textbf{repeat}\ f\ \ell'], s_{C[\textbf{repeat}\ f\ \ell']}$$

and the premise which allowed the rule to fire, namely:

$$C[f\ \ell], s_{C[f\ \ell]} \triangleright^k C[n \otimes \ell'], s_{C[n\otimes\ell']}$$

the desired result is achieved, since $s_{C[f\ \ell]} = s_{C[\textbf{repeat}\ f\ \ell]}$ and $s_{C[n\otimes\ell']} = s_{C[\textbf{repeat}\ f\ \ell']}$ using the uniqueness of $C[\textbf{repeat}\ f\ \ell']$ labels and Lemmas 40 and 41.

If $C[\textbf{repeat}\ f\ \ell']^o \not\rightarrow^*_{F_l} \underline{n}$, $C[\textbf{repeat}\ f\ \ell]^o \not\rightarrow^*_{F_l} \underline{n}$. By the induction hypothesis,

$$C[\textbf{repeat}\ f\ \ell'], s_{C[\textbf{repeat}\ f\ \ell']} \not\triangleright^* \underline{n}, s',\ any\ s'.$$

And, combining this with the argument above,

$$C[\textbf{repeat}\ f\ \ell], s_{C[\textbf{repeat}\ f\ \ell]} \not\triangleright^* \underline{n}, s',\ any\ s'.$$

The other case of **repeat** is quite similar to this case.

The cases where $C[\cdot]$ is **let** $x \otimes y$ **be** $M \otimes N$ **in** C, and $\pi_i\langle M_1, M_2\rangle$ are straightforward, as there is no change of state involved. ∎

Finally, we can give the proof of the main result.

Proof: [of Coincidence] Suppose that $\vdash_{\{\}} M{:}\,\mathbf{nat}$. By Lemmas 38 and 40, M has no location constants and is therefore a term in which each location occurs uniquely. Coincidence then follows immediately from Lemma 42. ∎

It is interesting to note that the proof gives more information than the result requires. The actual proof itself is a sort of "lock step" argument, where the reductions in the two semantics match up one by one.

4.3 No Garbage

Another property of the imperative operational semantics is that there is no garbage. That is, for any well typed term, there are no cells left at the end of the execution of a program. All must have been deallocated. This is particularly interesting as this language does not obey a stack discipline, in that cells can be allocated and deallocated in arbitrary order.

Theorem 43 (No Garbage) *If* $- \vdash^{\mathrm{imp}} M : \mathbf{nat}$ *and* $M, \{\} \triangleright^{*} \underline{n}, s$*, then* $s = \{\}$*.*

It is first necessary to prove a slightly stronger condition.

Lemma 44 *In the derivation* $M, s \triangleright^{*} \underline{n}, s'$*, if* $- \vdash_L M{:}\,\mathbf{nat}$*,* $\underline{n}{:}\,\mathbf{nat}$*,* $L = \mathit{freeloc}(M)$*, and* $s \in S(L)$*, then* $s' = \{\}$*, and* $\mathit{dom}(S) = L$*.*

Proof: This proof depends on Theorem 5 and Lemma 38. The proof is done by an induction on the length of the derivation. For each step in the derivation, there is a case analysis, based on the possible reduction rules that apply.

Consider rules of the form:

$$M = C[Q], s \triangleright C[Q'], s'$$

The induction hypothesis is:

$$C[Q'], s' \triangleright^* \underline{n}, \{\}$$

All of the cases are straightforward. Only one is shown.

Case: Q is $update(\underline{n'}, \ell)$. So,

$$C[update(\underline{n'}, \ell)], s \triangleright C[\ell], (s \mid \ell \mapsto \underline{n'}).$$

By the induction hypothesis,

$$C[\ell], (s \mid \ell \mapsto \underline{n'}) \triangleright^* \underline{n}, \{\}.$$

Since *update* preserves typings and locations, we are done. ■

Now the proposition follows immediately from Lemma 44.

Corollary 45 (Determinacy) *If* $- \vdash^{\text{imp}} M : \mathbf{nat}$, $M, \{\} \triangleright^* \underline{n}, s$, *and* $M, \{\} \triangleright^* \underline{n'}, s'$, *then* $\underline{n} = \underline{n'}$ *(and* $s = s' = \{\}$*).*

4.4 Call-By-Value Evaluation

In this section, we propose a call-by-value reduction strategy for the imperative operational semantics. At first glance, this seems an unusual strategy, as it would fail to be adequate for, say, call-by-name PCF, and we have used call-by-name to define our language. But, due to the fact that $\multimap$ is strict (or, analogously, that any variable being abstracted upon is unique), this strategy will be correct, as Theorem 46 will show. This point has been made before about linear λ-calculus by Abramsky [Abr93] and Braüner [Bra96] in the context of functional operational semantics. We indicate how our imperative semantics does not interfere with this observation.

Our reason for defining a reduction strategy is that we will need a defined reduction order for building a complexity model to analyze our depth-first search algorithm in Chapter 7. To define the reduction strategy, we use evaluation contexts [Fel87].

Evaluation contexts identify a unique subterm to be evaluated next. We define them as:

$$\begin{array}{rcl} E[\cdot] & = & [\cdot] \mid \textbf{let } x \otimes y \textbf{ be } E[\cdot] \otimes M \textbf{ in } M \mid \textbf{let } x \otimes y \textbf{ be } v \otimes E[\cdot] \textbf{ in } M \\ & & \mid E[\cdot](M) \mid (\lambda x.M)E[\cdot] \mid alloc(E[\cdot]) \mid update(E[\cdot], M) \\ & & \mid update(n, E[\cdot]) \mid lookup(E[\cdot]) \mid dealloc(E[\cdot]) \mid ifz(E[\cdot], M) \\ & & \mid \textbf{repeat } E[\cdot]\ M \mid \textbf{repeat } \lambda x.M\ E[\cdot] \mid +(E[\cdot] \otimes M) \\ & & \mid +(\underline{n} \otimes E[\cdot]) \mid \dot{-}\,(E[\cdot] \otimes M) \mid \dot{-}\,(\underline{n} \otimes E[\cdot]). \end{array}$$

Here, $v \in Val$, where $M, M' \in Val$ may be defined as follows:

$$M, M' \ ::= \ \underline{n} \mid \ell \mid M \otimes M' \mid \langle M, M' \rangle \mid \lambda x.M \mid \star$$

Now, we define an additional reduction relation, $\triangleright_E$, where the order of reductions is deterministic. This is in contrast to $\triangleright$ where the order of reductions may occur in any order. Figure 7 contains the definition of $\triangleright_E$. Here, for the case of **repeat**, $s' = s$ unless $(\lambda x.M)$ changes s.

The correctness of the call-by-value reduction strategy is stated in the following adequacy result.

Theorem 46 (CBV Adequacy) *If* $-\vdash_L^{\text{imp}} M{:}\,\textbf{nat}$, *then* $M, \{\} \triangleright_E^* n, s \iff M, \{\} \triangleright^* n, s$. *(Furthermore, s can always be taken to be $\{\}$, as all locations must have been deallocated).*

Proof: [Sketch] The proof of this proposition in the $\Longrightarrow$ direction is trivial. Since the reduction sequence under $\triangleright$ may occur in any sequence, it may occur in the sequence as specified by $\triangleright_E$. The proof of this proposition in the $\Longleftarrow$ direction is more difficult. It is possible to prove this directly by proving various linear properties about the type system. Instead, a more straightforward, brute-force approach is used. Only a broad outline of the proof is given here.

The proof is accomplished in three steps:

1. We define a functional, call-by-value, operational semantics in a manner analogous to our definition of the imperative call-by-value operational semantics. In

$$
\begin{array}{rcll}
(\lambda x.M)v, s & \triangleright_E & M[v/x], s & \\
\mathbf{let}\ x \otimes y\ \mathbf{be}\ v_1 \otimes v_2\ \mathbf{in}\ C, s & \triangleright_E & C[v_1/x, v_2/y], s & \\
\pi_i\langle M_1, M_2\rangle, s & \triangleright_E & M_i, s & \\
\mathbf{let}\ \star\ \mathbf{be}\ \star\ \mathbf{in}\ C, s & \triangleright_E & C, s & \\
\mathit{ifz}\,(\underline{0} \otimes M), s & \triangleright_E & \pi_1 M, s & \\
\mathit{ifz}\,(\underline{n} \otimes M), s & \triangleright_E & \pi_2 M, s \quad (\text{if } \underline{n} \neq \underline{0}) & \\
+\,(\underline{n} \otimes \underline{m}), s & \triangleright_E & \underline{n+m}, s & \\
\dot{-}\ (\underline{n} \otimes \underline{m}), s & \triangleright_E & \underline{n-m}, s \quad (\text{if } m < n) & \\
 & \triangleright_E & \underline{0}, s \quad \text{otherwise} & \\
\mathbf{repeat}\ (\lambda x.M)\ l, s & \triangleright_E^{(k+1)} & l', s \quad (\text{if } (\lambda x.M)\ l, s \triangleright_E^k \underline{0} \otimes l', s') & \\
 & \triangleright_E^{(k+1)} & \mathbf{repeat}\ (\lambda x.M)\ l', s' \quad (\text{if } (\lambda x.M)\ l, s \triangleright_E^k \underline{n} \otimes l', s') &
\end{array}
$$

$$
\frac{M, s \triangleright_E M', s'}{E[M], s \triangleright_E E[M'], s'}
$$

$$
\begin{array}{rcll}
\mathit{update}(\underline{n}, \ell),\ s & \triangleright_E & \ell,\ (s \mid \ell \mapsto \underline{n}) & (\ell \in \mathit{dom}\, s) \\
\mathit{lookup}(\ell),\ s & \triangleright_E & (\ell, s(\ell)),\ s & (\ell \in \mathit{dom}\, s) \\
\mathit{dealloc}(\ell),\ s & \triangleright_E & \star,\ s - \ell & (\ell \in \mathit{dom}\, s) \\
\mathit{alloc}(\underline{n}),\ s & \triangleright_E & \ell, (s \mid \ell \mapsto \underline{n}) & (\ell \notin \mathit{dom}\, s)
\end{array}
$$

Figure 7: Call-by-value imperative operational semantics

other words, we use evaluation contexts to specify the order of reduction, and we make minimal changes to the functional operational semantics reduction rules (similar to what was done for the imperative call-by-value reduction rules).

2. We prove a coincidence result connecting the functional call-by-value operational semantics to the imperative call-by-value operational semantics. The proof is almost identical to the proof of the coincidence result (Theorem 29). That proof implicitly established what is essentially a "lock step" result, showing how the semantics mimicked one another on the level of individual steps. It will be the same for the proof of the two call-by-value operational semantics.

3. We show that the functional call-by-value operational semantics is adequate with respect to the denotational $\mathbf{CPO}_\perp$ model. This adequacy between the functional call by value operational semantics and the $\mathbf{CPO}_\perp$ model is not as obvious as the adequacy between the initial functional operational semantics and the $\mathbf{CPO}_\perp$ model.

   The proof technique is similar to that done by Braüner [Bra96], who showed an adequacy result for the call-by-value semantics of linear logic, using a "many-step" formulation of reduction. While Braüner used a many-step formulation of reduction, we use a single-step formulation. However, similar ideas can be used here.

The results quoted in points 2 and 3 immediately imply the theorem. ■

In summary, we have shown adequacy between a functional interpretation of the language of Rudimentary Types and an imperative interpretation. However, the language of Rudimentary Types is rather weak. In the next two chapters, we move on to consider extensions to the language of Rudimentary Types (in the hopes of making it more powerful), and check whether the properties of subject reduction, adequacy with respect to a stateless denotational model, operational extensionality, and adequacy between a functional and an imperative interpretation of the language continue to hold.

Chapter 5

Adding “!”

This chapter considers extensions of the language of Rudimentary Types. The motivation for doing this is two-fold.

1. From a λ-calculus perspective, we hope to extend Rudimentary Types to develop a language with the full expressive power of the λ-calculus. Presently, the only function types $\theta \multimap \theta'$ are a strict subset of their λ-calculus equivalent, $\theta \rightarrow \theta'$.

2. From a programming perspective, we hope to extend Rudimentary Types to develop a language that has a generalized recursion mechanism. Additionally, we would like the resulting language to support objects, as commonly occurs when procedural abstraction is combined with assignment [Rey78, OT].

The extension added to the language is “!”. The intuition about !-types is that they may be used multiple times. The hope is that they can enable us to obtain the expressive power of the full λ-calculus. By adding new types, $!\theta$, we will need to add special reduction rules concerning “!”. Using these rules, which are standard, we will show that coincidence fails; that is, the functional and the imperative operational semantics no longer coincide. This leads us to consider a modified language, with an unusual constant, *block*, for allocating and deallocating storage variables. We show that coincidence does indeed hold for this new language. However, we find that operational extensionality now fails. We conclude the chapter by evaluating what the *block* construct does and does not solve.

This chapter confirms and adds to observations made by Wadler in [Wad91]. Since details of the proofs we give are similar to those presented in the previous chapters, we move fairly quickly over the material, indicating only the extensions to those proofs for the results claimed. Also, note that the main contributions of this chapter are not the proofs themselves, but rather the negative observations made by the failures of coincidence and operational extensionality.

5.1 "!" Types and Reductions

The grammar of types is changed to be the following:

$$\theta, \theta' \ ::= \ \varphi \mid I \mid \theta \otimes \theta' \mid \theta \times \theta' \mid \theta \multimap \theta' \mid !\theta$$

The intuition behind adding the new type $!\theta$ is that an expression of type $!\theta$ may be used multiple times in the evaluation of a larger expression in which it is occurs. Thus, a function of type $!\theta \multimap \theta'$ may use its argument multiple times. And, using Girard's translation, $\theta \rightarrow \theta' \equiv !\theta \multimap \theta'$ (note that $\theta \rightarrow \theta'$ is the set of continuous functions), all of the classical functions from the λ-calculus may be recovered.

With this new type, we also extend our collection of pre-terms, ranged over by M:

$$\begin{array}{rcl} M & ::= & x \mid k \mid \textbf{let } x \otimes y \textbf{ be } M_1 \textbf{ in } M_2 \mid M_1(M_2) \mid M_1 \otimes M_2 \\ & & \mid \pi_1 M \mid \pi_2 M \mid \langle M_1, M_2 \rangle \mid \lambda x.M_1 \mid \textbf{let } \star \textbf{ be } M_1 \textbf{ in } M_2 \\ & & \mid \textbf{promote } M_1, \cdots, M_n \textbf{ for } x_1, \cdots, x_n \textbf{ in } M \mid \star \\ & & \mid \textbf{derelict}(M) \mid \textbf{copy } z \textbf{ as } x, y \textbf{ in } M \mid \textbf{discard } M_1 \textbf{ in } M_2 \end{array}$$

The typing rules for this extended language are in Figure 8. They are taken from [BBdPH92].

The Promotion and Dereliction rules arise because, under certain circumstances, it is necessary to change the type of a term from $x\colon \theta$ to $x\colon !\theta$ and vice versa. Intuitively,

Rules from Figure 2, plus

$$\frac{\Delta_1 \vdash e_1\colon !\theta_1 \quad \cdots \quad \Delta_n \vdash e_n\colon !\theta_n \quad x_1\colon !\theta_1, \cdots, x_n\colon !\theta_n \vdash f\colon \theta}{\Delta_1, \cdots, \Delta_n \vdash \textbf{promote}\ e_1, \cdots, e_n\ \textbf{for}\ x_1, \cdots, x_n\ \textbf{in}\ f\colon !\theta}\ \textsf{Promotion}$$

$$\frac{\Delta \vdash t\colon !\theta}{\Delta \vdash \textbf{derelict}(t)\colon \theta}\ \textsf{Dereliction}$$

$$\frac{\Delta_1 \vdash z\colon !\theta' \quad \Delta_2, x\colon !\theta', y\colon !\theta' \vdash f\colon \theta}{\Delta_1, \Delta_2 \vdash \textbf{copy}\ z\ \textbf{as}\ x, y\ \textbf{in}\ f\colon \theta}\ \textsf{Contraction}$$

$$\frac{\Delta_1 \vdash e\colon !\theta' \quad \Delta_2 \vdash f\colon \theta}{\Delta_1, \Delta_2 \vdash \textbf{discard}\ e\ \textbf{in}\ f\colon \theta}\ \textsf{Weakening}$$

Figure 8: Typing rules for Linear Types with "!"

Dereliction says that if an expression is used multiple times, one of the ways it may be used is uniquely; Promotion states that if all of the assumptions of an expression (i.e., the set of variables that make up Δ) are non-unique, then the expression itself may be non-unique. The Weakening and Contraction rules are added as primitives (rather than as explicit function constants, as they had been for the language of Rudimentary Types), as it is always safe to discard or to duplicate a ! variable.

There are other approaches to typing rules for ! that have appeared in the literature, some of which interpret Contraction and Weakening more implicitly (the former allowing multiple occurrences of a variable). The differences between these type systems is not essential here.

The reductions for this λ-calculus are in Figure 9.

Rules from Figure 2, plus

$$
\begin{array}{rcl}
\textbf{derelict}\ (\textbf{promote}\ e_i\ \textbf{for}\ x_i\ \textbf{in}\ t) & \rightarrow & t[e_i/x_i] \\[1ex]
\textbf{discard}\ (\textbf{promote}\ e_i\ \textbf{for}\ x_i\ \textbf{in}\ t)\ \textbf{in}\ u & \rightarrow & \textbf{discard}\ e_i\ \textbf{in}\ u \\[1ex]
\textbf{copy}\ (\textbf{promote}\ e_i\ \textbf{for}\ z_i\ \textbf{in}\ t)\ \textbf{as}\ x, y\ \textbf{in}\ z & \rightarrow & \textbf{copy}\ e_i\ \textbf{as}\ z_i', z_i''\ \textbf{in} \\
& & u[\textbf{promote}\ z_i'\ \textbf{for}\ z_i\ \textbf{in}\ t/x, \\
& & \ \textbf{promote}\ z_i''\ \textbf{for}\ z_i\ \textbf{in}\ t/y]
\end{array}
$$

Figure 9: Standard reductions for Linear Types with "!"

5.2 The Sample Language, Revisited

The first language considered has exactly the same primitive types and constants as in Chapter 2. In particular, recall that we have four constants:

$$
\begin{array}{rclrcl}
alloc & : & \textbf{nat} \multimap \textbf{cell} & dealloc & : & \textbf{cell} \multimap I \\[1ex]
update & : & \textbf{nat} \otimes \textbf{cell} \multimap \textbf{cell} & lookup & : & \textbf{cell} \multimap \textbf{cell} \otimes \textbf{nat}
\end{array}
$$

The reduction rules $\rightarrow$ specified in Figure 9 induce functional and imperative operational semantics $\rightarrow_F$ and $\triangleright$, in exactly the same way as in Figure 3 and Figure 5 from Chapters 3 and 4.

5.2.1 Wadler's Problem

Wadler used graph reduction imagery in his discussion of linear λ-calculi. In graph reduction terms, ideally, the addition of ! would lead to a distinction between values of linear types, such as $\theta \multimap \theta'$, which have only only pointer to them, and values of non-linear types such as $!\theta$, which would allow multiple pointers.

Such an idea seems infeasible, however, because of an the interaction between Dereliction and Contraction. Given a value of linear type $!\theta$, a Contraction reduction may be performed to obtain two values of type $!\theta$, which may be tensored together to obtain a value of type $!\theta \otimes !\theta$. Subsequently, two Derelictions may be performed, to obtain a value of type $\theta \otimes \theta$. Interpreting Contraction by sharing and Dereliction by a no-op will break the above-stated property, because a value of type θ may have two pointers to it. A specific example used by Wadler to illustrate this is

$$\begin{array}{l} \lambda f \,.\, \lambda g \,.\, \lambda x \,.\, \mathbf{copy}\, x \,\mathbf{as}\, x_1, x_2 \,\mathbf{in}\, f(\mathbf{derelict}\, x_1) \otimes g(\mathbf{derelict}\, x_2) \\ : (\theta \multimap \theta') \multimap (\theta \multimap \theta'') \multimap\, !\theta \multimap (\theta' \otimes \theta'') \; . \end{array}$$

The functions f and g each take a linear argument. But, as a result of Dereliction, they may be passed a non-linear argument.

5.2.2 Failure of Coincidence

Wadler expressed his "problem" in terms of a graph semantics. We believe that it would not be difficult to formalize his observation, and similar conclusions have been reached in [CGR96].

In terms of our functional and imperative semantics, we make a remark that is directly related to destructive update.

Observation 47 *Coincidence fails.*

The reason coincidence fails is due to the Promotion and Dereliction rules. In the type system as it is presently defined, there is nothing to prevent the Promotion of a variable (or location), applying Contraction and then Dereliction to it, to produce a variable (or location) of non-! type which does *not* occur uniquely within a term. As uniqueness is critical for being able to destructively update a **cell**, we will now have broken coincidence. Specifically, let us reconsider the linear λ-calculus example from the Introduction (rewritten in this new language).

$$
\begin{array}{l}
(\lambda y.\lambda z.\lambda w. \\
\quad \mathbf{copy}\ y\ \mathbf{as}\ y', y''\ \mathbf{in} \\
\quad\quad \mathbf{let}\ val_1 \otimes z'\ \mathbf{be}\ lookup\ z\ \mathbf{in} \\
\quad\quad\quad \mathbf{let}\ val_2 \otimes w'\ \mathbf{be}\ lookup\ w\ \mathbf{in} \\
\quad\quad\quad\quad \mathbf{let}\ \star\ \mathbf{be}\ dealloc\ z'\ \mathbf{in} \\
\quad\quad\quad\quad\quad \mathbf{let}\ \star\ \mathbf{be}\ dealloc\ w'\ \mathbf{in} \\
\quad\quad\quad\quad\quad\quad + (val_1 \otimes val_2)) \\
\mathbf{promote}\ (alloc\ \underline{0})\ (update\ \underline{7}\ (\mathbf{derelict}\ y'))\ (update\ \underline{8}\ (\mathbf{derelict}\ y''))
\end{array}
$$

The example type checks in the new type system, and the functional result of evaluating this expression (which is 15) may or may not agree with the imperative result of evaluation (which depending on the order in which the evaluation occurs will be 14, 15, or 16). Note that the **promote** rule reduces to **promote** M when the contexts are empty.

For example, here is a reduction sequence which yields 16.

Initially, the evaluation of $(alloc\ \underline{0})$ occurs.

$$
\begin{array}{l}
(\lambda y.\lambda z.\lambda w. \\
\quad \mathbf{copy}\ y\ \mathbf{as}\ y', y''\ \mathbf{in} \\
\quad\quad \mathbf{let}\ val_1 \otimes z'\ \mathbf{be}\ lookup\ z\ \mathbf{in} \\
\quad\quad\quad \mathbf{let}\ val_2 \otimes w'\ \mathbf{be}\ lookup\ w\ \mathbf{in} \\
\quad\quad\quad\quad \mathbf{let}\ \star\ \mathbf{be}\ dealloc\ z'\ \mathbf{in} \\
\quad\quad\quad\quad\quad \mathbf{let}\ \star\ \mathbf{be}\ dealloc\ w'\ \mathbf{in} \\
\quad\quad\quad\quad\quad\quad + (val_1 \otimes val_2)) \\
\mathbf{promote}\ \ell\ (update\ \underline{7}\ (\mathbf{derelict}\ y'))\ (update\ \underline{8}\ (\mathbf{derelict}\ y'')), (s \mid \ell \mapsto \underline{0})
\end{array}
$$

Then, the evaluation of **promote** ℓ occurs. This is essentially a no-op, as no state change results.

$$
\begin{array}{l}
(\lambda y.\lambda z.\lambda w. \\
\quad \textbf{copy } y \textbf{ as } y', y'' \textbf{ in} \\
\quad\quad \textbf{let } val_1 \otimes z' \textbf{ be } lookup\ z \textbf{ in} \\
\quad\quad\quad \textbf{let } val_2 \otimes w' \textbf{ be } lookup\ w \textbf{ in} \\
\quad\quad\quad\quad \textbf{let } \star \textbf{ be } dealloc\ z' \textbf{ in} \\
\quad\quad\quad\quad\quad \textbf{let } \star \textbf{ be } dealloc\ w' \textbf{ in} \\
\quad\quad\quad\quad\quad\quad + (val_1 \otimes val_2)) \\
\ell\ (update\ \underline{7}\ (\textbf{derelict } y'))\ (update\ \underline{8}\ (\textbf{derelict } y'')), (s \mid \ell \mapsto \underline{0})
\end{array}
$$

Next, the first function application occurs.

$$
\begin{array}{l}
(\lambda z.\lambda w. \\
\quad \textbf{copy } \ell \textbf{ as } y', y'' \textbf{ in} \\
\quad\quad \textbf{let } val_1 \otimes z' \textbf{ be } lookup\ z \textbf{ in} \\
\quad\quad\quad \textbf{let } val_2 \otimes w' \textbf{ be } lookup\ w \textbf{ in} \\
\quad\quad\quad\quad \textbf{let } \star \textbf{ be } dealloc\ z' \textbf{ in} \\
\quad\quad\quad\quad\quad \textbf{let } \star \textbf{ be } dealloc\ w' \textbf{ in} \\
\quad\quad\quad\quad\quad\quad + (val_1 \otimes val_2)) \\
(update\ \underline{7}\ (\textbf{derelict } y'))\ (update\ \underline{8}\ (\textbf{derelict } y'')), (s \mid \ell \mapsto \underline{0})
\end{array}
$$

Then, the **copy** gets evaluated.

$$
\begin{array}{l}
(\lambda z.\lambda w. \\
\quad \textbf{let } val_1 \otimes z' \textbf{ be } lookup\ z \textbf{ in} \\
\quad\quad \textbf{let } val_2 \otimes w' \textbf{ be } lookup\ w \textbf{ in} \\
\quad\quad\quad \textbf{let } \star \textbf{ be } dealloc\ z' \textbf{ in} \\
\quad\quad\quad\quad \textbf{let } \star \textbf{ be } dealloc\ w' \textbf{ in} \\
\quad\quad\quad\quad\quad + (val_1 \otimes val_2)) \\
(update\ \underline{7}\ (\textbf{derelict } \ell))\ (update\ \underline{8}\ (\textbf{derelict } \ell)), (s \mid \ell \mapsto \underline{0})
\end{array}
$$

Next, the evaluation of both **derelict**s occurs (which does not result in any state change per se).

$$
\begin{array}{l}
(\lambda z.\lambda w. \\
\quad \mathbf{let}\ val_1 \otimes z' \ \mathbf{be}\ lookup\ z\ \mathbf{in} \\
\quad\quad \mathbf{let}\ val_2 \otimes w' \ \mathbf{be}\ lookup\ w\ \mathbf{in} \\
\quad\quad\quad \mathbf{let}\ \star\ \mathbf{be}\ dealloc\ z'\ \mathbf{in} \\
\quad\quad\quad\quad \mathbf{let}\ \star\ \mathbf{be}\ dealloc\ w'\ \mathbf{in} \\
\quad\quad\quad\quad\quad +\ (val_1 \otimes val_2)) \\
(update\ \underline{7}\ \ell)\ (update\ \underline{8}\ \ell), (s \mid \ell \mapsto \underline{0})
\end{array}
$$

Next, the evaluation of the next application occurs.

$$
\begin{array}{l}
(\lambda w. \\
\quad \mathbf{let}\ val_1 \otimes z' \ \mathbf{be}\ lookup\ (update\ \underline{7}\ \ell)\ \mathbf{in} \\
\quad\quad \mathbf{let}\ val_2 \otimes w' \ \mathbf{be}\ lookup\ w\ \mathbf{in} \\
\quad\quad\quad \mathbf{let}\ \star\ \mathbf{be}\ dealloc\ z'\ \mathbf{in} \\
\quad\quad\quad\quad \mathbf{let}\ \star\ \mathbf{be}\ dealloc\ w'\ \mathbf{in} \\
\quad\quad\quad\quad\quad +\ (val_1 \otimes val_2)) \\
(update\ \underline{8}\ \ell), (s \mid \ell \mapsto \underline{0})
\end{array}
$$

This is the key step. By overwriting the value of ℓ before the first *lookup* occurs, the first *lookup* will return a value of 8 instead of the value of 7 that is returned under the functional operational semantics. Now, the evaluation of the rest of the term occurs, essentially in any order. For argument's sake, we proceed as follows. The last application may occur next.

$$
\begin{array}{l}
\mathbf{let}\ val_1 \otimes z' \ \mathbf{be}\ lookup\ (update\ \underline{7}\ \ell)\ \mathbf{in} \\
\quad \mathbf{let}\ val_2 \otimes w' \ \mathbf{be}\ lookup\ (update\ \underline{8}\ \ell)\ \mathbf{in} \\
\quad\quad \mathbf{let}\ \star\ \mathbf{be}\ dealloc\ z'\ \mathbf{in} \\
\quad\quad\quad \mathbf{let}\ \star\ \mathbf{be}\ dealloc\ w'\ \mathbf{in} \\
\quad\quad\quad\quad +\ (val_1 \otimes val_2), (s \mid \ell \mapsto \underline{0})
\end{array}
$$

Evaluation of the first *update* occurs next.

$$\begin{array}{l}
\mathbf{let}\ val_1 \otimes z'\ \mathbf{be}\ lookup\ \ell\ \mathbf{in} \\
\quad \mathbf{let}\ val_2 \otimes w'\ \mathbf{be}\ lookup\ (update\ \underline{8}\ \ell)\ \mathbf{in} \\
\quad\quad \mathbf{let}\ \star\ \mathbf{be}\ dealloc\ z'\ \mathbf{in} \\
\quad\quad\quad \mathbf{let}\ \star\ \mathbf{be}\ dealloc\, w'\ \mathbf{in} \\
\quad\quad\quad\quad + (val_1 \otimes val_2), (s \mid \ell \mapsto \underline{7})
\end{array}$$

Next, the second *update* gets evaluated.

$$\begin{array}{l}
\mathbf{let}\ val_1 \otimes z'\ \mathbf{be}\ lookup\ \ell\ \mathbf{in} \\
\quad \mathbf{let}\ val_2 \otimes w'\ \mathbf{be}\ lookup\ \ell\ \mathbf{in} \\
\quad\quad \mathbf{let}\ \star\ \mathbf{be}\ dealloc\ z'\ \mathbf{in} \\
\quad\quad\quad \mathbf{let}\ \star\ \mathbf{be}\ dealloc\, w'\ \mathbf{in} \\
\quad\quad\quad\quad + (val_1 \otimes val_2), (s \mid \ell \mapsto \underline{8})
\end{array}$$

The first *lookup* evaluation occurs.

$$\begin{array}{l}
\mathbf{let}\ val_1 \otimes z'\ \mathbf{be}\ \underline{8} \otimes \ell\ \mathbf{in} \\
\quad \mathbf{let}\ val_2 \otimes w'\ \mathbf{be}\ lookup\ \ell\ \mathbf{in} \\
\quad\quad \mathbf{let}\ \star\ \mathbf{be}\ dealloc\ z'\ \mathbf{in} \\
\quad\quad\quad \mathbf{let}\ \star\ \mathbf{be}\ dealloc\, w'\ \mathbf{in} \\
\quad\quad\quad\quad + (val_1 \otimes val_2), (s \mid \ell \mapsto \underline{8})
\end{array}$$

The second *lookup* evaluation occurs.

$$\begin{array}{l}
\mathbf{let}\ val_1 \otimes z'\ \mathbf{be}\ \underline{8} \otimes \ell\ \mathbf{in} \\
\quad \mathbf{let}\ val_2 \otimes w'\ \mathbf{be}\ \underline{8} \otimes \ell\ \mathbf{in} \\
\quad\quad \mathbf{let}\ \star\ \mathbf{be}\ dealloc\ z'\ \mathbf{in} \\
\quad\quad\quad \mathbf{let}\ \star\ \mathbf{be}\ dealloc\, w'\ \mathbf{in} \\
\quad\quad\quad\quad + (val_1 \otimes val_2), (s \mid \ell \mapsto \underline{8})
\end{array}$$

The first **let** is evaluated.

$$\begin{array}{l}
\mathbf{let}\ val_2 \otimes w'\ \mathbf{be}\ \underline{8} \otimes \ell\ \mathbf{in} \\
\quad \mathbf{let}\ \star\ \mathbf{be}\ dealloc\ z'\ \mathbf{in} \\
\quad\quad \mathbf{let}\ \star\ \mathbf{be}\ dealloc\ w'\ \mathbf{in} \\
\quad\quad\quad + (val_1 \otimes val_2)[\underline{8}/val_1][z'/\ell], (s \mid \ell \mapsto \underline{8})
\end{array}$$

The evaluation of the second **let** occurs.

$$\begin{array}{l}\textbf{let } \star \textbf{ be } \mathit{dealloc}\, z' \textbf{ in} \\ \quad \textbf{let } \star \textbf{ be } \mathit{dealloc}\, w' \textbf{ in} \\ \quad\quad + (val_1 \otimes val_2)[\underline{8}/val_1][z'/\ell][\underline{8}/val_2][w'/\ell], (s \mid \ell \mapsto \underline{8})\end{array}$$

The two *dealloc*s occur.

$$\begin{array}{l}\textbf{let } \star \textbf{ be } \star \textbf{ in} \\ \quad \textbf{let } \star \textbf{ be } \star \textbf{ in} \\ \quad\quad + (val_1 \otimes val_2)[\underline{8}/val_1][\underline{8}/val_2], s\end{array}$$

The remaining **let**s get evaluated.

$$+ (val_1 \otimes val_2)[\underline{8}/val_1][\underline{8}/val_2], s$$

Finally, the addition occurs, resulting in a value of 16.

This is essentially the same sequence as given earlier, except that **derelict** and **copy** are done explicitly.

One might argue that the operational semantics, not linear typing, is to blame for this problem. However, the reductions used come directly from the standard categorical semantics of linear logic. This, together with the related observations in [Wad91, CGR96], makes us think that the problem is genuine and not an artifact of our chosen formalism.

5.3 The *block* language

One proposed solution to these difficulties with ! suggested by Wadler is to eliminate the *alloc* and *dealloc* function constants altogether, replacing them with a single constant *block*:

$$\mathit{block} : !(\mathbf{cell} \multimap \mathbf{cell} \otimes \mathbf{nat}) \multimap \mathbf{nat}.$$

The idea of *block* is that a **cell** gets allocated and then passed to a function which uses the **cell** in its computation, finally returning a **cell** paired with a **nat**. The

cell is then deallocated and the **nat** is returned to the caller. The allocation and deallocation of cells is done in a higher-order fashion to prevent them from being duplicated or discarded.

In graph reduction terms, the intuition for why this should work is that, since **cell** is not of the form $!\theta$, one should know that the number of pointers to a cell will never increase. Further, the *block* constant is a special mechanism for creating a cell with only one pointer to it. The higher-order nature of *block* is also supposed to prevent uses of Promotion, which would allow creation of a cell with multiple pointers to it.

In our terms, it is enough to ask simply whether coincidence holds. The rest of this section involves defining the imperative and functional operational semantics for the resulting language, outlining the proof of coincidence, and confirming (in our terms) the argument made by Wadler. At the end of the section we observe a limitation to the language, however, in that operational extensionality fails.

5.3.1 Functional and Imperative Operational Semantics

In this section, we consider a language in which the *alloc* and *dealloc* constants are omitted and replaced by the *block* constant (as described above).

For the purpose of the functional operational semantics, the constant *alloc* (technically, the labeled allocs) is kept. So, there are actually two languages, one of which is only of interest in formulating the operational semantics.

Definition 48 *We write* $\Gamma \vdash M : \theta$ *for a derivable judgement in the language with the block constant, but no alloc or dealloc.*

Definition 49 *We write* $\Gamma \vdash^a M : \theta$ *for a derivable judgement in the language with the block and alloc constants, but no dealloc.*

We will formulate our results, such as subject reduction, for the $\vdash^a$ language. They will be meaningful for $\vdash$ as well because of the following lemma.

Lemma 50 (Conservativity) *If alloc does not appear in* M, $\Gamma \vdash M : \theta \iff \Gamma \vdash^a M : \theta$.

Proof: Straightforward. ■

The functional operational semantics for *lookup* and *update* remain exactly as in Figure 3. The semantics of *block* is as follows:

$$\frac{(\mathbf{derelict}\, f)(alloc\, \underline{0}) \rightarrow_F^k \ell \otimes \underline{n}}{block\, f \rightarrow_F^{(k+1)} \underline{n}}$$

Like previously, we use the terminology $\rightarrow_F^k$ to mean "reduces in k steps", $\rightarrow_F$ to mean "reduces in 1 step", and $\rightarrow_F^*$ to mean "reduces in 0 or more steps".

Note that the corresponding analogs of the first free variable proposition (Proposition 1) and cut elimination theorem (Proposition 4) for the language of Rudimentary Types continue to hold for the language with *block*. Additionally, the uniqueness properties, as expressed in Definition 2 and Proposition 3, continue to hold, because Contraction and Weakening are explicit.

We can now state the subject reduction theorem:

Proposition 51 (Subject Reduction for *block* functional operational semantics) *If* $\Delta \vdash^a t\colon \theta$ *and* $t \rightarrow_F^* u$, *then* $\Delta \vdash^a u\colon \theta$.

Proof: Like the previous subject reduction proofs, the proof is by an induction over the length of a derivation of a term t. For both the base and inductive cases, there is a further case analysis, considering each of the rules that could have been used in the reduction of the term, t. It is quite similar to the proof of Proposition 5, from Chapter 3. The rules for "!" have been considered elsewhere [BBdPH92]. The only other new rule is the one for *block*.

Case: t is reduced by

$$\frac{(\mathbf{derelict}\ f)\ (alloc\, \underline{0}) \rightarrow_F^k \ell \otimes \underline{n}}{block\ f \rightarrow_F^{(k+1)} \underline{n}}$$

Here, it is necessary to show that $\Delta \vdash_{L'}^{\mathrm{imp}} \underline{n}\colon \mathbf{nat}$ if $\Delta \vdash_{L}^{\mathrm{imp}} block\ f : \mathbf{nat}$.

Since (**derelict** f) ($alloc$ $\underline{0}$) reduces to $\ell \otimes \underline{n}$ via a strictly shorter derivation (namely in k steps), we have that $\Delta \vdash_{L''}^{\mathrm{imp}} \ell \otimes \underline{n}\colon \mathbf{cell} \otimes \mathbf{nat}$. By uniqueness, we know that $\Delta = \{\}$. Now, since $\Delta = \{\}$, we know that $- \vdash^{\mathrm{imp}}_{L'} (\mathbf{derelict}\ f) \colon \mathbf{cell} \multimap \mathbf{cell} \otimes \mathbf{nat}$, and that $- \vdash^{\mathrm{imp}}_{L} f \colon \mathbf{cell} \multimap \mathbf{cell} \otimes \mathbf{nat}$ since **derelict** is just a function constant. And since *block* is just a function constant, we have that $- \vdash^{\mathrm{imp}}_{L'''} \mathit{block}\ f \colon \mathbf{nat}$, which is what we needed to show, since $- \vdash^{\mathrm{imp}}_{\{\}} \underline{n} \colon \mathbf{nat}$ because n is just a constant.

■

The imperative operational semantics is specified following Figure 5 in Chapter 4. A new definition of $\rightarrow$ is given in this chapter, and from that the imperative semantics $\triangleright$ is obtained. The only change from the rules in Figure 5 is that the rules for *alloc* and *dealloc* are removed and the following rule for *block* is added:

$$\frac{f\ \ell, (s \mid \ell \mapsto \underline{0}) \triangleright \ell \otimes \underline{n}, s'}{\mathit{block}\ f, s \triangleright^{*} \underline{n}, s' - \ell}$$

Subject reduction for the imperative operational semantics for the *block* language can now be stated. We first note that the propositions involving locations (Propositions 32 and 33) continue to hold for this language.

Proposition 52 (Subject Reduction for *block* imperative operational semantics) *If $\Delta \vdash^{\mathrm{imp}}_{L} t \colon \theta$ and $t, s \triangleright^{*} u, s'$, then there exists an L' such that $\Delta \vdash^{\mathrm{imp}}_{L'} u \colon \theta$.*

Proof: The case of *block* is quite similar to the *block* case for the functional subject reduction of Proposition 51. ■

5.3.2 Coincidence

Before proving coincidence, some lemmas that are analogous to those we proved in Section 4.2.3 first need to be proved.

Lemma 53 *For well-typed terms in the imperative operational semantics of the block language, every reduction rule except for block and* **repeat** *preserves the set of locations. In other words, if $\Delta \vdash^{\mathrm{imp}}_{L} t \colon \theta$, $t, s \triangleright u, s'$ (and the reduction that occurs does NOT involve block or* **repeat***), then $\Delta \vdash^{\mathrm{imp}}_{L} u \colon \theta$.*

It is worthwhile to make a comment about the **copy** rule, as it would seem possible to copy a location, breaking this lemma. But this is not the case, as there are no terms of type **cell** in the language. The use of the *block* construct is especially designed to only allow **cell**s at higher order.

Lemma 54 *For well-typed terms in the functional operational semantics of the block language, every reduction rule except for block and* **repeat** *preserves the set of locations. In other words, if* $t\colon\theta$ *is well typed, if* $\Delta \vdash^{f}_{L} t\colon\theta$, $t \rightarrow_F u$ *(and the reduction that occurs does NOT involve block or* **repeat***), then* $\Delta \vdash^{f}_{L} u\colon\theta$.

Lemma 55 *For well-typed terms, no reduction rule in the imperative semantics for the block language duplicates locations. In other words, if* $\Delta \vdash^{\mathrm{imp}}_{L} t\colon\theta$, $t, s \triangleright^{*} u, s'$, *and* $\Delta' \vdash^{\mathrm{imp}}_{L'} u\colon\theta'$, *if* $l \in L$ *and* l *appears uniquely in* t, *then* l *appears either uniquely, or not at all in* u.

Lemma 56 *In the functional operational semantics of the block language, no labeled allocs (i.e., locations) are duplicated in any well typed term.*

We finally prove a slightly stronger condition, of which coincidence is a special case.

Lemma 57 *Suppose that* M *is a term with location constants in which each location occurs uniquely. Suppose that* s *is a state, and that* $freeloc(M) \subseteq dom(s)$. *Then,* $\exists s'.M, s \triangleright^{*} \underline{n}, s' \iff M^{o} \rightarrow^{*}_{F} \underline{n}$ *where* $M^{o} = M$ *with* $(alloc_l(\underline{n})/\ell)$.

Proof: The proof technique is the same as was done for the language of Rudimentary Types. Most of the cases are identical. We only work through the *block* case in detail.
Case: $C[block\ f]^{o} \rightarrow_F C[\underline{m}]^{o}$.

Since $C[\underline{m}]^{o} \rightarrow^{*}_{F} \underline{n}$ has a shorter derivation length,

$$C[\underline{m}] \rightarrow^{*}_{F} \underline{n} \iff \exists s'.C[\underline{m}], s_{C[\underline{m}]} \triangleright^{*} \underline{n}, s'$$

There are two possible subcases:

If $C[\underline{m}]^o \rightarrow_F^* \underline{n}$,

$$C[block\ f]^o \rightarrow_{F_l} C[\underline{m}]^o \rightarrow_F^* \underline{n}$$

So, using the induction hypothesis,

$$C[\underline{m}], s_{C[\underline{m}]} \triangleright^* \underline{n}, s'$$

combined with

$$C[block\ f], s_{C[block\ f]} \triangleright C[\underline{m}], s_{C[\underline{m}]-\ell}$$

and the premise which allowed the rule to fire, namely:

$$C[f\ell], s_{C[f\ell]} \mid \ell \mapsto \underline{0} \triangleright^k C[\ell \otimes \underline{m}], s_{C[\ell \otimes \underline{m}]}$$

the desired result is achieved, since $s_{C[block f]} = s_{C[f\ell]} \mid \ell \mapsto \underline{0}$, and $s_{C[\underline{m}]} - \ell = s_{C[\underline{m}]}$ using the uniqueness of $C[\underline{m}]$ labels and Lemmas 55 and 56.

If $C[\underline{m}]^o \not\rightarrow_F^* \underline{n}$, $C[block\ f]^o \not\rightarrow_F^* \underline{n}$. By the induction hypothesis,

$$C[\underline{m}], s_{C[\underline{m}]} \not\triangleright^* \underline{n}, s',\ any\ s'.$$

And, combining this with the argument above,

$$C[block\ f], s_{C[block\ f]} \not\triangleright^* \underline{n}, s',\ any\ s'.$$

■

And, immediately as a result of Lemma 57, we get:

Theorem 58 (Coincidence) *If* $- \vdash M : \mathbf{nat}$ *then*

$$M \rightarrow_F^* \underline{n} \iff \exists s\,(M, \{\} \triangleright^* \underline{n}, s)$$

5.3.3 Functional Properties

The material in this section consists of mostly negative results. We only briefly comment on the results, omitting proofs.

A Denotational Model

The denotational model given in Section 3.2 can be extended to the larger fragment of linear type theory by interpreting ! using lifting. For the new constant, we can define the function

$$block : (N_\perp \multimap N_\perp \otimes N_\perp) \rightarrow N_\perp$$

to simply be the function taking f to $f(0); \Pi_2$, where Π_2 is the evident second projection. Although this gives a model, it is not a very good model, as we now indicate.

Failure of Operational Extensionality

As mentioned previously, the block constant appears to be unusual from the point of view of functional programming. In particular, it violates operational extensionality.

Observation 59 *Operational extensionality fails. That is, for* $- \vdash M : \theta_1 \multimap \theta_2$ *and* $- \vdash N : \theta_1 \multimap \theta_2$ *it is not the case that* $- \vdash M \sqsubseteq N$ *if and only if, for all* $- \vdash V : \theta_1$, $- \vdash MV \sqsubseteq NV$.

We will not give a formal proof of this (which would be very involved), but will simply say why it is so. First, the formulation of operational extensionality would extend the one earlier where, in particular, two closed terms of function type are equivalent if their applications to closed arguments are all equivalent. Then consider the functions $\lambda c : \mathbf{cell}\,.\, update\, \underline{7}\, c$ and $\lambda c : \mathbf{cell}\,.\, update\, \underline{8}\, c$. Certainly these terms are not equivalent, but there are no closed terms of type **cell** on which they differ. In fact, it is not difficult to see that, up to contextual equivalence, there are no closed terms of type **cell** in the language, other than divergent terms (assuming we add a divergent term $\Omega_{\mathbf{cell}}$ of type **cell**).

Observation 60 *If* $- \vdash M : \mathbf{cell}$ *then* $M \equiv \Omega$, *where* $\equiv$ *is the evident contextual equivalence relation.*

So, to distinguish $\lambda c : \mathbf{cell}\,.\, update\, \underline{7}\, c$ and $\lambda c : \mathbf{cell}\,.\, update\, \underline{8}\, c$, it is necessary to go beyond applicative contexts, using for instance:

$$C[\cdot] \quad = \quad block\,!(\lambda\, c : \mathbf{cell}\,.\, lookup([\cdot]c)).$$

This failure of operational extensionality is similar to the situation in Idealized Algol [O'H95].

This points to a large mismatch with the denotational model. In the model, there are infinitely many maps $I \multimap N$ corresponding semantically to what judgements $- \vdash M\colon \mathbf{cell}$ should be. But there are no closed terms of type **cell**, other than ones that denote $\perp$.

A Functor Category Model

A better denotational model can be obtained by using a functor category to model the local-variable nature of *block*. Such a model is only very briefly sketched. This follows work on the semantics of Idealized Algol. Here, we use a very simple functor category to illustrate the main points.

The category we use is the arrow category $(\mathbf{CPO}_\perp)^\rightarrow$. That is, it is a functor category, where the category of possible worlds $\rightarrow$ has two objects and one non-identity arrow between them. We picture the category of worlds as:

$$global \xrightarrow{\quad insert \quad} local$$

$(\mathbf{CPO}_\perp)^\rightarrow$ inherits important categorical structure from $\mathbf{CPO}_\perp$. It is a symmetric monoidal closed category, with finite products, and an appropriate co-monad obtained from lifting. This gives the structure to model linear type theory. In particular, the exponent of functors is defined on objects so that

$$(F \multimap G)x \;=\; (\mathbf{CPO}_\perp)^{\rightarrow}[I(x,\cdot)_\perp \otimes F, G]$$

where $I(x,\cdot)_\perp$ is the evident lifting of the representable functor and $\mathbf{CPO}_\perp^{\rightarrow}[\cdot,\cdot]$ denotes the hom set in the functor category (equipped with the pointwise order). The tensor, $\otimes$, is defined on objects so that $(F \otimes G)x = Fx \otimes Gx$.

The semantics of the primitive types is such that $[\![\mathbf{nat}]\!]$ is the constant functor where $[\![\mathbf{nat}]\!]x = N_\perp$, and cells are given by $[\![\mathbf{cell}]\!]global = \{\perp\}$, $[\![\mathbf{cell}]\!]local = N_\perp$, and $[\![\mathbf{cell}]\!]insert$ is the unique strict function. The idea is that there are indeed cells to use in a local context, inside a *block*, and their contents are natural numbers, but that there are no cells in a global context.

The semantics of *block* can then be given by changing possible worlds to allow an extra cell, just as in the semantics of local variables in imperative languages. The definition of the natural transformation

$$block : [\![\mathbf{cell} \multimap \mathbf{cell} \otimes \mathbf{nat}]\!] \dot{\to} [\![\mathbf{nat}]\!]$$

is given by

$$\begin{array}{rcl} block[global]f & = & f(insert, 0); \Pi_1 \\ block[local]f & = & f(id, 0); \Pi_1 \end{array}$$

The semantic role of *block* is to insert a global function into a local context. The semantics of other terms are straightforward.

Now, with this semantics the following observation can be made.

Observation 61 *There is only one natural transformation* $[\![I]\!] \dot{\to} [\![\mathbf{cell}]\!]$*, the everywhere-*$\perp$ *map.*

We expect to model a closed term of type θ by a map $[\![I]\!] \dot{\to} [\![\theta]\!]$ in the category $(\mathbf{CPO}_\perp)^{\to}$. So this observation indicates that the semantics in one respect captures the "local/global" aspect of the block language better than the $\mathbf{CPO}_\perp$ model.

A formal proof of the adequacy of the functor model with respect to the operational semantics has not been carried out, but it appears that such a verification would be routine. More interesting is that the functor model suggests an alternate, "possible worlds" form of operational extensionality that should hold for the language. Such a formulation of extensionality would take into account the global/local distinction, or even the number of cells allocated at any one time. The extensionality clause for function types would follow the functor-category exponent closely.

This alternate form of extensionality will not be given here. Although such a form of extensionality could be useful, the additional use of worlds only serves to further emphasize the non-functional characteristic of *block*. Further, this possible worlds extensionality we have in mind is a variation on the one shown by Pitts for Idealized Algol [Pit96].

5.4 Where are we now

From a programming perspective, some of our goals in the language with *block* may be achieved. For example, it is trivial to add a recursion combinator

$$Y \colon !(\,!\theta \multimap \theta) \multimap \theta,$$

with reduction rule

$$Y(f) \rightarrow f\,Y(f).$$

This addition would allow us to finally define recursive functions, including functions with multiple recursive calls in the function body, in a language satisfying coincidence.

Our other programming motivation was to have a reasonable *interaction* between imperative and functional features. In Idealized Algol, for example, classes can be defined using procedures and local variable declarations. If P is a procedure identifier of type $\mathbf{comm} \times \mathbf{exp} \rightarrow \mathbf{comm}$ (it takes a command and an expression as arguments, and produces a command), then

$$\mathbf{new}\; x \,.\, x := 0;\; P(x := x + 1, x)$$

declares a new "counter object" for use within P. λ-abstracting on P results in a "counter class".

Attempts to capture this kind of behavior in the *block* language have failed. The reason is that, in a block $block(\lambda c \cdots P \cdots)$, if P is to interact with c, it must get hold of the whole cell, which it can then manipulate in arbitrary, non-counter-like, ways. We do not have a proof that this kind of treatment of objects cannot be done in the *block* language (the formulation, let alone proof, of such a result would be difficult), but a way to do it is far from clear. From an imperative programming point of view, we find the *block* language to be rather weak (as we do not see a way to add objects and classes to the language).

Finally, the failure of operational extensionality for the language with *block* is somewhat disappointing. Despite the fact that a "possible worlds" form of extensionality probably does hold, the added requirement of quantifying extensionality over "possible worlds" is not the general form of operational extensionality for which we were hoping. This seems to indicate a non-purely-functional aspect of the language.

This failure of operational extensionality leads us to look for other extensions to Rudimentary Types. In the next chapter, Wadler's steadfast types are examined.

Chapter 6

Steadfast Types

In this chapter, we consider Wadler's second solution to the problem of coincidence failing when !-types are added to the language of Rudimentary Types (or any other such similar fragment of intuitionistic linear logic). We will formulate Steadfast Types, and then argue that a problem that Wadler claims to solve still exists, by presenting some troubling examples. The examples are based on an encoding of an alternative "!" into steadfast types, which in effect enables the embedding of the whole of (intuitionistic) linear logic into steadfast types. One of Wadler's original examples is displayed, reworked for this encoding, and it is shown that coincidence fails for this language. The difficulty observed does not appear to be due to the specifics of our operational models. Indeed, the "one-pointer property" claimed for steadfast types in [Wad91] appears to fail. We then re-formulate Steadfast Types, and show that the new language, Steadfast Types Revisited, does maintain a coincidence property, as well as having a good denotational model, and having operational extensionality hold. We conclude the chapter by examining what benefit Steadfast Types Revisited has over the language of Rudimentary Types.

6.1 Wadler's Steadfast Types

The intuition behind Steadfast Types is fairly straightforward. Wadler notices that the problem of linear variables having multiple pointers — which is what leads intuitively to the difficulties noted in Sections 5.2.1 and 5.2.2 — is essentially the result of the use of the Promotion and Dereliction rules. A linear variable (such as a **cell**) may be promoted to a !-type, after which Contraction may be applied to the variable; when the variable is finally derelicted, it returns to being a linear variable, but with multiple pointers. In such cases, it is no longer safe to do destructive update.

Since Dereliction and Promotion are at the root of the problem, Wadler proposes to simply eliminate these rules. However, in order to be able to do any reductions with terms of !-type, Wadler adds $\times$ and $\rightarrow$ to the type system, as non-linear analogs of $\otimes$ and $\multimap$.

6.1.1 The Type System

There are two presentations of Steadfast Types, one as a subsystem of linear logic, with a restricted "!", and one as a system without "!" but with a distinction drawn between linear and non-linear types. The second form of presentation is used here, which is as in [Wad90] and mentioned as a "syntactic trick" in [Wad91]. Nothing set forth is sensitive to this choice of presentation.

Our illustrative version of Steadfast Types will have two primitive types, **nat** and **cell**. The types are as follows:

$$\theta, \theta' \quad ::= \quad \mathbf{nat} \mid \mathbf{cell} \mid I \mid \theta \otimes \theta' \mid \theta \multimap \theta' \mid \theta \times \theta' \mid \theta \rightarrow \theta'$$

Definition 62 *The following defines linear and non-linear types, and non-linear contexts:*

1. Linear Types*:* **cell***,* I *and types of the form* $\theta \multimap \theta'$ *and* $\theta \otimes \theta'$ *are called linear types.*

2. Nonlinear Types*:* **nat** *and types of the form* $\theta \to \theta'$ *and* $\theta \times \theta'$ *are called nonlinear types.*

3. Nonlinear Contexts*: A context* $\Gamma = x_1 : \theta_1, \ldots, x_n : \theta_n$ *is called nonlinear if each* θ_i *is a nonlinear type.*

It is perhaps debatable whether I should be classified as linear or nonlinear.

The type system will select from a collection of pre-terms, defined identically to the set of pre-terms defined for the language of Rudimentary Types in Chapter 2. It would be conceivable to distinguish syntactically between binding and application forms for $\multimap$ and $\to$, but this will not be done.

Figure 10 consists of the typing rules for Steadfast Types. It is worth making a brief remark on the overall collection of typing rules, as well as commenting about some of the typing rules in particular. Wadler's Steadfast Typing rules do not have Promotion or Dereliction. Rather, Dereliction is implicit in the elimination rule for $\theta \to \theta'$ (reading it as $!(\theta \multimap \theta'))$, and promotion is implicit in the introduction rule. One can also regard $\times$ and **nat** as being "implicitly !'d." However, this view of nonlinear types as being !'d is not completely accurate, because the resulting Dereliction and Promotion operations are more restrictive than in linear logic. Note also that the expression "Δ nonlinear" means that all of the identifiers in Δ are themselves nonlinear.

Wadler claimed that Steadfast Types resulted in a system with the "one-pointer property." Recall that this property says that, in a graph reduction semantics, variables of linear type always have one pointer associated with them, whereas variables of nonlinear type may have multiple pointers.

6.1.2 The Problem: An Alternate Encoding for "!"

Recall from Section 5.2.1 that Wadler showed a difficulty with the "one pointer property" in a language with full !. Here we show that one can encode an alternate ! in steadfast types, from which a similar difficulty appears to follow.

$$\frac{}{\iota\colon\theta \vdash \iota\colon\theta}\ \text{Axiom} \qquad\qquad \frac{}{- \vdash k\colon type(k)}\ \text{Constant}$$

$$\frac{\Delta \vdash t\colon\theta}{\Delta, a\colon\theta' \vdash t\colon\theta}\ \textsf{Weakening}\ (\text{when } \theta' \text{ is nonlinear})$$

$$\frac{\Delta, x\colon\theta', y\colon\theta' \vdash t\colon\theta}{\Delta, z\colon\theta' \vdash\ t[z/x, z/y]\colon\theta}\ \textsf{Contraction}\ (\text{when } \theta' \text{ is nonlinear })$$

$$\frac{}{- \vdash \star\colon I}\ \star I \qquad\qquad \frac{\Delta_1 \vdash t\colon I \quad \Delta_2 \vdash u\colon\theta}{\Delta_1, \Delta_2 \vdash \mathbf{let}\,\star\ \mathbf{be}\ t\ \mathbf{in}\ u\colon\theta}\ \star E$$

$$\frac{\Delta \vdash P\colon\theta_1 \quad \Delta \vdash Q\colon\theta_2}{\Delta \vdash \langle P, Q\rangle\colon\theta_1 \times \theta_2}\ \times I \qquad\qquad \frac{\Delta \vdash P\colon\theta_1 \times \theta_2}{\Delta \vdash \pi_i\, P\colon\theta_i}\ \times E_i\ (i = 1, 2)$$

$$\frac{\Delta_1 \vdash P\colon\theta_1 \quad \Delta_2 \vdash Q\colon\theta_2}{\Delta_1, \Delta_2 \vdash P \otimes Q\colon\theta_1 \otimes \theta_2}\ \otimes I \qquad\qquad \frac{\Delta_1, x_1\colon\theta_1, x_2\colon\theta_2 \vdash t\colon\theta_3 \quad \Delta_2 \vdash u\colon\theta_1 \otimes \theta_2}{\Delta_1, \Delta_2 \vdash \mathbf{let}\ x_1 \otimes x_2\colon\theta_1 \otimes \theta_2\ \mathbf{be}\ u\ \mathbf{in}\ t\colon\theta_3}\ \otimes E$$

$$\frac{\Delta, \iota\colon\theta' \vdash P\colon\theta}{\Delta \vdash \lambda\iota\colon\theta'.P\colon\theta' \to \theta}\ {\to}I\ (\Delta \text{ nonlinear}) \qquad\qquad \frac{\Delta_1 \vdash P\colon\theta' \to \theta \quad \Delta_2 \vdash Q\colon\theta'}{\Delta_1, \Delta_2 \vdash P(Q)\colon\theta}\ {\to}E$$

$$\frac{\Delta, \iota\colon\theta' \vdash P\colon\theta}{\Delta \vdash \lambda\iota\colon\theta'.P\colon\theta' \multimap \theta}\ {\multimap}\, I \qquad\qquad \frac{\Delta_1 \vdash P\colon\theta' \multimap \theta \quad \Delta_2 \vdash Q\colon\theta'}{\Delta_1, \Delta_2 \vdash P(Q)\colon\theta}\ {\multimap}\, E$$

Figure 10: Typing rules for Steadfast Types

The problem involves the mixing of $\rightarrow$ and $\multimap$ within a term. In Steadfast Types, "Each type is deemed linear or nonlinear, depending on its topmost constructor. Hence $(T \multimap (U \rightarrow V))$ is linear, while $(T \rightarrow (U \multimap V))$ is nonlinear." [Wad90] The problem is with non-linear functions of type $(T \rightarrow (U \multimap V))$. A function of this type is non-linear (and may have $\mathsf{Contraction}$ or $\mathsf{Weakening}$ performed against it), but once it gets applied to an argument, it "becomes" linear.

Formally, the problem arises as follows: We define a type constructor $\Box A$ as syntactic sugar for $I \rightarrow A$. Now, the following operations may be defined in the language:

$$\begin{array}{rcl} \mathsf{Dereliction} & : & \Box A \multimap A \\ \mathsf{Contraction} & : & \Box A \multimap \Box A \otimes \Box A \\ \mathsf{Weakening} & : & \Box A \multimap I \\ \mathsf{Promotion} & : & (\Box A \multimap B) \multimap (\Box A \multimap \Box B) \end{array}$$

In particular, we now get versions of $\mathsf{Promotion}$ and $\mathsf{Dereliction}$ back into the language of SteadfastTypes. The troublesome linear λ-calculus example from the Introduction now type checks, illustrating that coincidence does *not* hold for this language.

The definitions of these terms are

$$\begin{array}{rcl} \mathsf{Dereliction} & = & \lambda f \,.\, f(*) \\ \mathsf{Contraction} & = & \lambda f \,.\, f \otimes f \\ \mathsf{Weakening} & = & \lambda f \,.\, * \\ \mathsf{Promotion} & = & \lambda f \,.\, \lambda a \,.(\lambda x \,.\, f(a)) \end{array}$$

Notice the use of Weakening of x in $\mathsf{Promotion}$, Contraction of f in $\mathsf{Contraction}$, and Weakening of f in $\mathsf{Weakening}$. These are well typed because I and $I \rightarrow A$ are nonlinear.

Now, Wadler's example from Section 5.2.1 can be typed as follows:

$$\begin{array}{l} \lambda f \,.\, \lambda g \,.\, \lambda x \,.\, \mathbf{let}\, x_1 \otimes x_2 \,\mathbf{be}\, \mathsf{Contraction}\ x\ \mathbf{in}\ \langle f(\mathsf{Dereliction}\ x_1), g(\mathsf{Dereliction}\ x_2)\rangle \\ : (\theta \multimap \theta') \multimap (\theta \multimap \theta'') \multimap \Box\theta \multimap (\theta' \otimes \theta'')\ . \end{array}$$

We believe that exactly the same informal reasoning used by Wadler against the "one-pointer property" for linear types with ! can now be made against steadfast types, with $\Box$ in place of !. Making this rigorous would involve a diversion into formulating an appropriate graph model. We will be content, therefore, to leave this example as is, and instead make a related observation in terms of our imperative operational semantics.

Before continuing several other remarks should be made. First, if I were not in the language, it would still be possible to formulate a version of the $\Box$ modality, using $\theta \multimap \theta'$ for any non-linear type, θ. For example, it would be possible to use $(\theta \to \theta) \to \theta'$, and then use $\lambda x.x$ in place of $*$ in the definitions of the maps. Second, the symbol $\Box$ was used, instead of !, to avoid confusion with Wadler's view of Steadfast Types as a subsystem of linear logic. For this purpose, Wadler has a different ! in mind than $\Box$.

6.1.3 Failure of Coincidence

To this language of Steadfast Types, the function constants *alloc*, *dealloc*, *lookup*, and *update* are added. In [Wad90], Wadler gives slightly different types to analogous constants, using $\times$ instead of $\otimes$. The comments about to be made would go through for either typing for the constants.

A variation of the failure of coincidence example from Chapter 5 is still typable, as illustrated here:

$$
\begin{array}{l}
(\lambda y.\lambda z.\lambda w.\\
\quad \mathbf{let}\ y' \otimes y''\ \mathbf{be}\ \mathsf{Contraction\ y}\ \mathbf{in}\\
\quad\quad \mathbf{let}\ val_1 \otimes z'\ \mathbf{be}\ lookup\ z\ \mathbf{in}\\
\quad\quad\quad \mathbf{let}\ val_2 \otimes w'\ \mathbf{be}\ lookup\ w\ \mathbf{in}\\
\quad\quad\quad\quad \mathbf{let}\ \star\ \mathbf{be}\ dealloc\ z'\ \mathbf{in}\\
\quad\quad\quad\quad\quad \mathbf{let}\ \star\ \mathbf{be}\ dealloc\ w'\ \mathbf{in}\\
\quad\quad\quad\quad\quad\quad +\ (val_1 \otimes val_2))\\
\Box(alloc\ \underline{0})\ (update\ \underline{7}\ (\mathsf{Dereliction}\ y'))\ (update\ \underline{8}\ (\mathsf{Dereliction}\ y''))
\end{array}
$$

Here, $\Box(alloc\ \underline{0})$ is used as syntactic sugar for **promote** $(\lambda x{:}\,I.(alloc\ \underline{0}))$. And, like before, this example will evaluate to 16 in the imperative operational semantics, while it will evaluate to 15 in the functional operational semantics.

6.2 Steadfast Types Revisited

Although Wadler's language of Steadfast Types, in its details, appears to have difficulties, Wadler's basic idea of restricting Dereliction and Promotion, or bundling them into new connectives, is an interesting one that seems to be promising. In this section, these ideas are examined in further detail, to develop a language in which coincidence does hold. After proving that the resulting language does have the coincidence property, an examination is made as to what it offers in terms of extending the language of Rudimentary Types.

6.2.1 Preliminaries

The Steadfast Types system had difficulties with types of the form $(T \to (U \multimap V))$, so a subtle, but significant change is made to the system of types. Types $\delta, \delta', \gamma, \gamma'$ are defined to be the following:

$$\begin{array}{rcll} \delta, \delta' & ::= & \varphi \mid \delta \otimes \delta' \mid \delta \multimap \delta' \mid I \mid \gamma & \text{Linear Types} \\ \gamma, \gamma' & ::= & \rho \mid \gamma \times \gamma' \mid \gamma \to \gamma' & \text{Non-Linear Types} \end{array}$$

Here, φ ranges over some set of primitive linear types (in our example languages, φ is simply **cell**), and ρ ranges over some set of primitive non-linear types (in our example languages, ρ is **nat**.) The intent of devising this two-tiered type system is to, in general, forbid types of the form $(T \to (U \multimap V))$. When typing rules for this Steadfast Types Revisited language are presented, it will become apparent how this is achieved.

Pre-terms can be defined just as they have been for Rudimentary Types. A slightly different notation for the presentation of typing judgements will be used, following the notation of Barber [Bar96]. Rather than using the normal one-zone approach,

$$\Delta \vdash M\colon \theta,$$

$$\frac{}{\Gamma, \iota{:}\,\theta; - \vdash \iota{:}\,\theta}\ \text{Non-Linear Vars} \qquad \frac{}{\Gamma; \iota{:}\,\theta \vdash \iota{:}\,\theta}\ \text{Linear Vars}$$

$$\frac{}{\Gamma; - \vdash k{:}\, type(k)}\ \text{Constant}$$

$$\frac{}{\Gamma; - \vdash \star{:}\, I}\ \star I \qquad \frac{\Gamma; \Delta_1 \vdash t{:}\, I \quad \Gamma; \Delta_2 \vdash u{:}\,\theta}{\Gamma; \Delta_1, \Delta_2 \vdash \mathbf{let}\,\star\ \mathbf{be}\ t\ \mathbf{in}\ u{:}\,\theta}\ \star E$$

$$\frac{\Gamma; \Delta \vdash P{:}\,\theta_1 \quad \Gamma; \Delta \vdash Q{:}\,\theta_2}{\Gamma; \Delta \vdash \langle P, Q\rangle{:}\,\theta_1 \times \theta_2}\ \times I \qquad \frac{\Gamma; \Delta \vdash P{:}\,\theta_1 \times \theta_2}{\Gamma; \Delta \vdash \pi_i\, P{:}\,\theta_i}\ \times E_i\ (i = 1, 2)$$

$$\frac{\Gamma; \Delta_1 \vdash P{:}\,\theta_1 \quad \Gamma; \Delta_2 \vdash Q{:}\,\theta_2}{\Gamma; \Delta_1, \Delta_2 \vdash P \otimes Q{:}\,\theta_1 \otimes \theta_2}\ \otimes I \qquad \frac{\Gamma; \Delta_1, x_1{:}\,\theta_1, x_2{:}\,\theta_2 \vdash t{:}\,\theta_3 \quad \Gamma; \Delta_2 \vdash u{:}\,\theta_1 \otimes \theta_2}{\Gamma; \Delta_1, \Delta_2 \vdash \mathbf{let}\ x_1 \otimes x_2{:}\,\theta_1 \otimes \theta_2\ \mathbf{be}\ u\ \mathbf{in}\ t{:}\,\theta_3}\ \otimes E$$

$$\frac{\Gamma, \iota{:}\,\theta'; \Delta \vdash P{:}\,\theta}{\Gamma; \Delta \vdash \lambda\iota{:}\,\theta'.P{:}\,\theta' \to \theta}\ \to I \qquad \frac{\Gamma; \Delta \vdash P{:}\,\theta' \to \theta \quad \Gamma; - \vdash Q{:}\,\theta'}{\Gamma; \Delta \vdash P(Q){:}\,\theta}\ \to E$$

$$\frac{\Gamma; \Delta, \iota{:}\,\theta' \vdash P{:}\,\theta}{\Gamma; \Delta \vdash \lambda\iota{:}\,\theta'.P{:}\,\theta' \multimap \theta}\ \multimap I \qquad \frac{\Gamma; \Delta_1 \vdash P{:}\,\theta' \multimap \theta \quad \Gamma; \Delta_2 \vdash Q{:}\,\theta'}{\Gamma; \Delta_1, \Delta_2 \vdash P(Q){:}\,\theta}\ \multimap E$$

Figure 11: Typing rules for Steadfast Types Revisited

where all free identifiers $x{:}\,\theta'$ are in Δ, a two-zone approach is used,

$$\Gamma; \Delta \vdash M{:}\,\theta$$

where the context is broken into an intuitionistic zone, Γ (identifiers of non-linear type, γ, are in Γ), and a linear zone, Δ (identifiers of linear type, δ, are in Δ).

Figure 11 consists of a first version of typing rules for the the modification of Steadfast Types. We make some important remarks concerning several of the typing rules. There is no need for explicit Contraction and Weakening rules: they are both allowed implicitly for identifiers in the Γ zone. There are now two identifier rules — one for non-linear identifiers and one for linear (technically, not non-linear) identifiers.

$$\frac{\Gamma; \Delta_1 \vdash P\colon \theta' \multimap \theta \quad \Gamma; \Delta_2 \vdash Q\colon \theta'}{\Gamma; \Delta_1, \Delta_2 \vdash P(Q)\colon \theta} \text{ if } \theta' \text{ is not non-linear or } \Delta_2 = \{\}$$

Figure 12: Corrected $\multimap E$ reduction

It is important to note that, for $\rightarrow$E, the linear zone for the argument *must* be empty. Otherwise, it would be possible to contract linear free identifiers that occur within the argument.

This seems like the most obvious collection of typing rules but, even though types that cause specific difficulties in Steadfast Types have been removed, a subtle flaw remains: The rule for $\multimap$ E is, as it currently stands, not correct. It would be possible for the type of the argument, θ' to be non-linear, and if Δ_2 were non-empty, it would be possible to contract linear identifiers (namely, those in Δ_2). For example, if a function were to take a non-linear argument and contract it, any locations used to make up the argument would get duplicated. Thus, a similar program to the one presented in Section 6.1.3 would be typable. Obviously, this would break coincidence.

As a result, our attempts to prove coincidence eventually lead us to a different elimination rule for $\multimap$. It is given in Figure 12. Intuitively, the restriction in the rule may be justified as follows. Because linear types can "contain" non-linear "pieces", it is necessary to ensure that the argument part of the potentially non-linear piece does not, itself, contain any linear sub-pieces. If it did, the troublesome examples from above would resurface.

However, it must be said that this version of the elimination rule is uncomfortably ad hoc. The coincidence property is surprisingly delicate, in that it is easy to formulate faulty type systems (which show themselves in the failure of coincidence) beyond the satisfactory system of rudimentary types. As a result, we do not believe that a satisfactory extension to the language of Rudimentary Types has been found, if one is concerned with mixing functional and imperative programming. While there are some systems that appear to work correctly from a technical point of view (here,

Rules from Figure 3, plus

$$Y(f) \quad \rightarrow_F \quad f\,Y(f)$$

Figure 13: β-reductions for Steadfast Types Revisited - functional operational semantics

in [GH90], in [BS93], etc.), none of the type systems are as elegant as the language of Rudimentary Types (in which coincidence holds, but the language is weak) or as the language of Linear Types with ! (in which coincidence fails, but the language is more expressive than the language of Rudimentary Types). Having admitted our discomfort, we nevertheless move on to consider properties of Steadfast Types Revisited.

6.2.2 Semantics and Properties of the Language

First we note that the uniqueness and free variable propositions claimed to hold for the language of Rudimentary Types (Propositions 1, 3, and 4) continue to hold here. This language may be extended with the four array function constants, *alloc*, *dealloc*, *lookup*, and *update*, whose types are defined in Chapter 2, and whose operational behavior is defined below.

In this section, the functional operational semantics of this language are defined. The three important functional criteria are shown: that reductions may occur in any order, that the language has a (stateless) denotational model and is adequate with respect to that model, and that operational extensionality holds for the language.

Functional Operational Semantics

Figure 13 contains all of the β-reduction rules for defining the functional operational semantics for the modification of Wadler's Steadfast Types. The reductions are identical to the reductions in the functional operational semantics for the language of

Rudimentary Types, with the exception of the recursion operator. The type of Y is:

$$Y\colon(\gamma \rightarrow \gamma) \rightarrow \gamma$$

Subject Reduction - Functional

Proposition 63 (Subject Reduction for functional operational semantics) *If* $\Gamma; \Delta \vdash t\colon\theta$ *and* $t \rightarrow_F^* u$, *then* $\Gamma; \Delta \vdash u\colon\theta$.

Proof: The proof is virtually identical to the proof of subject reduction for the functional operational semantics of the language of Rudimentary Types in Chapter 3. The only new case is the following:

Case: t is $Y(f)$. Thus, t must have had a derivation as follows:

$$\frac{\Gamma; \Delta \vdash f\colon\gamma \rightarrow \gamma \quad \Gamma; - \vdash Y\colon(\gamma \rightarrow \gamma) \rightarrow \gamma}{\Gamma; \Delta \vdash Y(f)\colon\gamma}$$

Now, because of the $\rightarrow$E rule requirements, $\Delta = \{\}$. So, another $\rightarrow$E can be applied, resulting in the desired result.

$$\frac{\Gamma; - \vdash f\colon\gamma \rightarrow \gamma \quad \Gamma; - \vdash Y(f)\colon\gamma}{\Gamma; - \vdash f\ Y(f)\colon\gamma}$$

It is also worthwhile to comment on the $\multimap E$ rule. It is more "restrictive" than the $\multimap E$ rule from Chapter 3, so subject reduction follows for it in a straightforward manner. ■

A Denotational Model

The denotational model used for the language of Rudimentary Types (defined in Section 3.2) can be used, exactly as it is defined, to model the modified Steadfast Types language, with the following extension. It is necessary to model $\theta \rightarrow \theta'$, which can be done as:

$$[\![\theta \rightarrow \theta']\!] = \text{continuous functions from } [\![\theta]\!] \text{ to } [\![\theta']\!]$$

Rules from Figure 5, plus

$$Y(f), s \qquad \triangleright \quad f\, Y(f), s$$

Figure 14: β-reductions for Steadfast Types Revisited - imperative operational semantics

The statement and proof of adequacy between the denotational model and the functional operational semantics are also the same (there is only the additional case of recursion, which, like the rest of the proof, is standard).

Operational Extensionality

Like the denotational model, the proof of operational extensionality is virtually identical to the proof presented in Section 3.3. The added definition for the recursion reduction needs to be added into the definition of $\leq^*$, and the recursion case has to be added for many of the propositions and lemmas, but it does not present any difficulties to the proofs.

Imperative Operational Semantics

Figure 14 contains all of the β-reduction rules for defining the imperative operational semantics for the modification of Wadler's Steadfast Types. The reductions are identical to the reductions in the imperative operational semantics for the language of Rudimentary Types, with the exception of the newly added recursion operator.

Subject Reduction - Imperative

It is worthwhile to note that the analogs of Propositions 32 and 33 continue to hold for the imperative operational semantics for this language. Subject reduction is stated as follows:

Proposition 64 (Subject Reduction for imperative operational semantics) *If* $\Gamma; \Delta \vdash_L^{\mathrm{imp}} t\colon \theta$ *and* $t, s \rhd^* u, s'$, *then* $\exists L'.\Gamma; \Delta \vdash_{L'}^{\mathrm{imp}} u\colon \theta$.

Proof: The proof is practically identical to the proof of Theorem 34. The additional case for recursion is not interesting as there is no state change involved. ■

Coincidence

The proof of coincidence between the functional and imperative operational semantics is very similar to the proof in Section 4.2.3. The main lemmas are restated and differences noted.

We start by extending the language as is done in Definitions 35 and 36, and note that Proposition 37 still holds. The following lemmas are needed, which guarantee that locations (linear identifiers) do not get duplicated in a harmful way.

Lemma 65 *For well-typed terms in the imperative operational semantics, every reduction rule except for alloc, dealloc, and* **repeat** *preserves the set of locations. In other words, if* $\Gamma; \Delta \vdash_L^{\mathrm{imp}} t\colon \theta$, $t, s \rhd u, s'$ *(and the reduction that occurs does NOT involve alloc, dealloc, or* **repeat***), then* $\Gamma; \Delta \vdash_L^{\mathrm{imp}} u\colon \theta$.

Lemma 66 *For well-typed terms in the functional operational semantics, every reduction rule except for alloc, dealloc, and* **repeat** *preserves the set of locations. In other words, if* $t\colon \theta$ *is well typed, if* $\Gamma; \Delta \vdash_L^{f} t\colon \theta$, $t \rightarrow_{F_l} u$ *(and the reduction that occurs does NOT involve alloc, dealloc, or* **repeat***), then* $\Gamma; \Delta \vdash_L^{f} u\colon \theta$.

The proofs of these lemmas are similar to the corresponding proofs in Section 4.2.3. The stringent requirement on the $\multimap$-E case ensures that these lemmas hold for this case as well.

Lemma 67 *For well-typed terms, no reduction rule in the imperative semantics duplicates locations. In other words, if* $\Gamma; \Delta \vdash_L^{\mathrm{imp}} t\colon \theta$, $t, s \rhd u, s'$, *and* $\Gamma'; \Delta' \vdash_{L'}^{\mathrm{imp}} u\colon \theta'$, *if* $\ell \in L$ *and* ℓ *appears uniquely in* t, *then* ℓ *appears either uniquely, or not at all in* u.

Lemma 68 *In the functional operational semantics, no labeled allocs (i.e., locations) are duplicated in any well typed term.*

Note that the $\multimap$-E case holds because of the stringent requirements on the type of the argument, and the free identifiers within it, and that the $\rightarrow$-E case holds because there may be no free linear identifiers within the argument.

Finally, we are able to prove a statement that is stronger than coincidence, from which coincidence is an immediate result.

Lemma 69 *Suppose that M is a term with location constants in which each location occurs uniquely. Suppose that s is a state, and that $freeloc(M) \subseteq dom(S)$. Then, $\exists s'. M, s \triangleright^* \underline{n}, s' \iff M^o \rightarrow^*_{Fl} \underline{n}$ where $M^o = M$ with $(alloc_l(\underline{n})/\ell)$.*

Again, the proof is very similar to the proof in Section 4.2.3.

6.2.3 Recursion

Presently, the recursive function constant for this language has type $(\gamma \rightarrow \gamma) \rightarrow \gamma$. (Remember that γ, as presented in Section 6.2.1, is of non-linear type.) While this function constant is fine for non-linear functions, it would be nice to have a recursive function constant for those functions that take arguments of linear type, as well. It seems useful to allow those recursive functions that operate on linear types (such as arrays). The problem with adding such a function constant can be expressed both intuitively as well as actually. Intuitively, linear terms (including linear functions) must be used exactly once. A function that takes an argument of linear type is itself necessarily of linear type as well. A recursive function needs some sort of "base case", where a value is simply returned rather than another call being made to the recursive function. This base case will not use the linear function, so that this breaks our intuitions about being used once. Additionally, some recursive functions have multiple recursive calls. In actuality, allowing the possible Weakening of a linear function (or of any linear expression, for that matter), breaks Theorem 43, our "no garbage" theorem. (This theorem, while not proved for this Steadfast Types Revisited language, does continue to hold. The proof for the theorem is quite similar to that which is proved in Section 4.3.) Moreover, recursive functions that contain multiple recursive calls (such as *quicksort*, which is defined below), may potentially break coincidence as they require Contraction on a linear type (namely on themselves!).

Perhaps the modification to allow *Weakening* would not be so bad. It would be possible to redefine the notion of *uniqueness* to account for the fact that the particular, unique variable could occur *not at all* within a term. It would then be possible to change the reduction rules to allow variables within the Δ zone to be weakened (in much the same manner as we presently allow Weakening for those variables within the Γ zone). Our property of "no garbage" will be discarded in the next chapter anyway, which will be done for complexity reasons.

However, even with this modification, it is still not possible to type many recursive functions. Those recursive functions that require multiple recursive calls still present a problem, as Contraction is not allowed on linear identifiers. As an example, consider *quicksort*, which may be loosely defined as:

$$
\begin{aligned}
\mathit{quicksort} \equiv \quad & (\lambda l.\ \mathbf{let}\ n \otimes l'\ \mathbf{be}\ \mathit{length}\ l\ \mathbf{in} \\
& \quad \mathit{ifz}\,(n \otimes \langle l', \mathit{ifz}\,(\dot{-}\,(n \otimes 1) \otimes \langle l', \\
& \quad\ \ \mathbf{let}\ x \otimes l''\ \mathbf{be}\ \mathit{pickpivot}\ l'\ \mathbf{in} \\
& \quad\ \ \ \ \mathbf{let}\ y \otimes l'''\ \mathbf{be}\ \mathit{rearrange}\ (l'', x)\ \mathbf{in} \\
& \quad\ \ \ \ \ \ \mathbf{let}\ l_1''' \otimes l_2'''\ \mathbf{be}\ \mathit{Break}\ (l''', y)\ \mathbf{in} \\
& \quad\ \ \ \ \ \ \ \ \mathit{Merge}\ (\mathit{quicksort}\ l_1'''), (\mathit{quicksort}\ l_2''')\ \rangle\rangle))
\end{aligned}
$$

In order to type *quicksort*, the *Break* and *Merge* functions would need to be added as primitives into the language. *Break* takes an n-element array and breaks it into two smaller ones, and *Merge* takes two smaller arrays and combines them into one array. (Dynamic arrays would need to be supported.) But the overall problem is the double occurrence of the *quicksort* function. Clearly, *quicksort*, which is of type $\mathbf{cell} \multimap \mathbf{cell}$, may *not* be contracted. However, *quicksort is* an eminently reasonable function, which should be typable in the language. The reason *quicksort* should be typable is that it would have the same imperative and functional behavior.

Other type systems, such as that of [GH90], seem to return a possible error (the algorithm of [GH90] appears to return multiple write, indicating a potential error) when trying to type *quicksort*.

It seems possible to allow a limited form of Contraction in the typing rules. For example, we could allow *Contraction* to the $\otimes$-I rule as follows:

$$\frac{\Gamma; \Delta_1 \vdash qs\ l \colon \theta \quad \Gamma; \Delta_2 \vdash qs\ l' \colon \theta}{\Gamma; \Delta_1, (\Delta_2 - qs) \vdash qs\ l \otimes qs\ l' \colon \theta \otimes \theta}$$

if the following conditions are met:

1. $(\Delta_1 - qs) \cup (\Delta_2 - qs) = \{\}$.
2. $qs \colon \theta' \multimap \theta$.
3. $Numocc(\theta') \neq fail$. And,
4. $Numocc(\theta') = Numocc(\theta)$.

Here, $Numocc(\theta)$ refers to the number of occurrences of linear types within the type, and may be defined inductively as indicated in Table 1. But this mechanism adds a huge amount of complexity to the typing rules, and makes the typing rules seem more like an algorithm.

6.2.4 What Steadfast Types Revisited Does and Does Not Do

This modification to Wadler's Steadfast Type system is somewhat disappointing. The type system becomes significantly more complex than that for the simple language of Rudimentary Types. It is true that we have been able to add a recursion function constant, but it is only applicable for terms (functions) of non-linear type. We would like to have a recursion function constant of type $Y \colon (\delta \to \delta) \to \delta$. Attempting to allow the recursion constant to admit linear functions results in breaking the "no garbage" property of the language, and trying to extend the constant to handle more

Type of θ	$Numocc(\theta)$
cell	1
nat	0
I	0
$\theta' \otimes \theta''$	$Numocc(\theta) + Numocc(\theta')$
$\theta' \times \theta''$	$Numocc(\theta')$ if $Numocc(\theta') = Numocc(\theta'')$
	$fail$ otherwise
$\theta' \rightarrow \theta''$	$Numocc(\theta')$ if $Numocc(\theta') = Numocc(\theta'')$
	$fail$ otherwise
$\theta' \multimap \theta''$	$Numocc(\theta')$ if $Numocc(\theta') = Numocc(\theta'')$
	$fail$ otherwise

Table 1: Calculation of number of occurrences of linear types

forms of recursion leads to a much greater complexity in the typing rules. We still have not been successful in our attempt to add objects to the language (there is no mechanism in the language to handle local state). So, overall, we are not so sure that Steadfast Types Revisited provides much of an advantage over simple Rudimentary Types.

Chapter 7

Depth First Search

In this chapter, the language of Rudimentary Types is again considered. A cost model for this language is introduced, and an informal justification for cost assignment is provided. To illustrate the cost model, we show that an implementation of Tarjan's depth-first search algorithm for graphs runs in time $O(v + e)$, where v and e are the numbers of vertices and edges in the graph to be searched. This is the same asymptotic run time that depth-first search has in standard imperative languages. A critical feature of the implementation of this algorithm in standard imperative languages is the use of destructive array updates to keep track of which nodes in the graph have been visited as the algorithm traverses the graph. The presence of safe destructive update in the language of Rudimentary Types, as guaranteed by the coincidence result, allows us to take advantage of this critical feature of standard imperative imperative implementations but in a functional language! In fact, the algorithm is not conceptually difficult to program in the language of Rudimentary Types.

Depth-first search has traditionally been considered to be a thorn in the side of functional programming. Previously, there has not been a linear depth-first search in a purely functional language. King and Launchbury [KL95] do consider depth-first search in an extension of Haskell, but they do not provide a complexity analysis (relying instead on measurements), and it would be quite a stretch to regard the version they use of Haskell as purely functional. This last point may be argued, since the term "purely functional" is ill defined and often used differently by different people. What is meant here is that it is not readily apparent how the monad used in their version of Haskell may be understood in terms of functional notions such as values, except by allowing state-to-state functions as values. So, it seems that states would likely be required to understand the semantics, just as in an imperative language. Their version of Haskell in fact appears to be in many ways reminiscent of Idealized Algol.

This chapter is divided into five sections. The first section concerns adding multi-element arrays and their associated function constants to the language. Reduction rules for these new constants are provided. In the second section a cost model for the language is introduced with costs assigned to each of the possible reduction steps. The third section consists of the depth-first search algorithm and an analysis of its run time under our cost model. The fourth section discussed the relative power of Fortran77 compared to the language of Rudimentary Types. Finally, the last section discusses omissions and additional work that needs to be done.

7.1 Adding Multi-element Arrays to the Language of Rudimentary Types

7.1.1 Multi-element Array Constants

In the language of Rudimentary Types, the only function constant for allocating arrays is *alloc*, and it only allocated a single cell array. Here the language is extended with an infinite collection of function constants:

$$alloc_k \quad : \quad \mathbf{nat} \multimap \mathbf{array}_k$$

where k is a natural number and $\mathbf{array}_k$ is a new type. The intent is that $alloc_k$ produces an array of k *consecutive* cells, each initialized to the value passed in the argument. This approach to arrays is Pascal-like, in that there are no dynamic arrays: in $\mathbf{array}_k$, k must be a constant and not a variable. The other operations are as follows.

$$\begin{array}{rcl} lookup_k & : & \mathbf{array}_k \otimes \mathbf{nat} \multimap \mathbf{array}_k \otimes \mathbf{nat} \\ update_k & : & \mathbf{nat} \otimes \mathbf{array}_k \otimes \mathbf{nat} \multimap \mathbf{array}_k \\ dealloc_k & : & \mathbf{array}_k \multimap I \end{array}$$

In the types for $lookup_k$ and $update_k$, the **nat** arguments (the second **nat** argument, for the case of $update_k$) are for indexing into the array.

One might have thought it would have been possible to define arrays inductively, as follows:

$$\begin{array}{rcl} \mathbf{array}_1 & \equiv_{def} & \mathbf{cell} \\ \mathbf{array}_{k+1} & \equiv_{def} & \mathbf{array}_k \otimes \mathbf{cell} \end{array}$$

Then $lookup_k$ and the other operations could have been programmed analogously in the language. In fact, one can define isomorphisms between these inductively-defined types and the array types described above. As a result, it might seem that the language extension being considered is not an extension at all, at least from the point of view of observational equivalence. However, observational equivalence does not take complexity into account. For the inductively defined array types, the evident candidate operations for $lookup_k$ and $update_k$ would take time proportional to k, because it would be necessary to traverse through the cells searching for the correct one. There is no reason that the actual machine on which this language is implemented should allocate the **cell**s consecutively in the computer's memory. The net result is that an **array** would be a linked list of individual **cell**s. The necessary resulting complexity model would make the complexity of the depth-first search algorithm $O(v \times (v + e))$. This complexity would not be acceptable, as a linear-time algorithm is desired.

It is also possible to add multi-dimensional arrays, but they are just syntactic sugar for one-dimensional arrays. In other words,

$$alloc_{k \times l} \equiv alloc_{k,l}$$

where $alloc_{k,l}$ is a 2-dimensional array with k rows and l columns. Both notations are freely used through the rest of this work, depending on which is easier to read/understand.

In contrast to the inductively-defined arrays, this encoding of two-dimensional arrays in terms of one-dimensional arrays is standard in algorithms work and is acceptable from a complexity-theoretic point of view. Other complexity assumptions are discussed further below.

7.1.2 Added Reduction Rules

To help specify a complexity model for this language a deterministic order of evaluation needs to be given. The particular evaluation strategy chosen is the call-by-value semantics, described in Section 4.4. The evaluation contexts need to be modified, to handle the new constants. This enforces a deterministic order in their evaluation. The new evaluation contexts are:

$$\begin{array}{rcl} E & = & \cdots \mid alloc_k(E) \mid update_k(E, M, N) \mid update_k(n, E, N) \\ & & \mid update_k(\underline{n}, \ell, E) \mid lookup_k(E, M) \mid lookup_k(\ell, E) \mid dealloc_k(E). \end{array}$$

Because arrays are now present in the language, it is necessary to slightly modify the notion of state. A state is now a finite function, $s\colon N \hookrightarrow N$, from the natural numbers to the natural numbers. The following reductions must be added for the new function constants:

$$\begin{array}{rcll} alloc_k(\underline{n}), s & \triangleright_E & \ell, (s \mid \ell \mapsto \underline{n}, \cdots, \ell + (k-1) \mapsto \underline{n}) & (\ell, \cdots, \ell + (k-1) \notin dom\ s) \\ \\ update_k(\underline{n}, \ell, i), s & \triangleright_E & \ell, (s \mid (\ell + i) \mapsto \underline{n}) & (\ell + i \in dom\ s) \wedge i < k \\ & \triangleright_E & \Omega_{\mathbf{array}_k} & otherwise \\ \\ lookup_k(\ell, i), s & \triangleright_E & (\ell, s(\ell + i)), s & (\ell + i \in dom\ s) \wedge i < k \\ & \triangleright_E & \Omega_{\mathbf{array}_k \otimes \mathbf{nat}} & otherwise \\ \\ dealloc_k(\ell), s & \triangleright_E & \star, s & \end{array}$$

Array offsets have typical side conditions in these rules, which should be viewed as a run-time safety check.

To claim a coincidence result connecting the imperative and functional operational semantics of this new language, it is necessary to give a new proof of the coincidence theorem. As these proofs are virtually identical to the previous proofs, details are not provided.

Note that *dealloc* is now interpreted as a "no-op". This is done to simplify complexity considerations, which are discussed below. This does not affect the proof of coincidence, but the "no garbage" result fails with this interpretation.

7.2 A Cost Model for the Language of Rudimentary Types

The complexity model on which an analysis of the language of Rudimentary Types is based is motivated by the RAM model. The model is a deterministic one, which is well suited for modeling the costs associated with running programs in an imperative language.

7.2.1 The RAM model

Random Access Machines (RAMs) are described in detail in [Pap94]. We simply summarize the main ideas. There is also a detailed description of a cost model for a simple imperative language based on the RAM model in [Jon97], from which much of our discussion is motivated. A RAM is a machine that contains a group of registers (used for loading data and addresses and for storing the results of mathematical computations), together with a large memory. The model contains an instruction set (for writing programs), which includes instructions for loading data into and out of memory, branching instructions, a test instruction, etc.

To use the RAM model as the basis for realistic analysis of the time complexity of algorithms, it is necessary to make some important assumptions on the data. For the purpose of the depth-first search algorithm, two that are crucial are:

1. All values being added together are assumed to be sufficiently small so that the $+$ and $*$ operations require only unit time.

2. The memory used in a program is assumed to be sufficiently small so that any memory location can be updated or read from in unit time.

These assumptions are informal. In particular, a fixed upper-bound on the size of numbers is not specified because otherwise algorithms would take constant time. Other (perhaps more standard) RAM assumptions that are made include the following:

1. The RAM model presented here allows for indirect addressing.

2. The RAM model presented here assumes that the lookup into an indexed array takes $O(1)$, just like the lookup into any other memory location.

In comparison, a Turing Machine may take time linear in space complexity to access a particular location, although variations on the standard Turing Machine model reduce this to time logarithmic in the space complexity. The unit-time assumptions are regarded as convenient and useful for the analysis of a range of algorithms. They are critical for getting a linear-time depth-first search.

7.2.2 Specific Assumptions of Our Model

In terms of array operations, the assumptions made above are adopted. The justifications for these assumptions are the same as they are for the general RAM model, together with the coincidence result showing that update can be done destructively in our language. We add two additional assumptions that are specific to our depth-first search algorithm.

1. Our model assumes that sufficient free space can be found for allocating an array of size k in unit time, so that $alloc_k$ really only costs k.

2. Our model treats $dealloc_k$ as a null operation.

The last assumption is somewhat disappointing. As mentioned previously, it clearly breaks the "no garbage" property previously discussed. There appears to be a tradeoff between the complexity of $alloc_k$ and $dealloc_k$. For instance, if the storage from an array were to be reclaimed after a $dealloc_k$ (remembering that we are using a heap discipline), it would be necessary to keep some sort of "free list" of pointers to freed space. The time then required for $alloc_k$ would depend on the structure of the free list, and might require more than unit time for finding enough contiguous free space. (The amount of time required will depend on the program which has already been executed.)

This issue of reclaiming storage after it is no longer needed is a standard issue in compilers. Many texts, including [AG97], discuss the issues of garbage collection in great detail. Here, this last point is only briefly explained by means of an example. For the sake of argument, the computer's memory is assumed to be of size n and each memory location i (note that $1 \leq i \leq n$) is identified by M_i. At the start of a program, the free memory looks like:

$$M_1, \ldots, M_n.$$

After the allocation of an array of size n_1, the free memory looks like:

$$M_{n_1+1}, \ldots, M_n,$$

and the free memory pointer list contains a single entry which points to memory element M_{n_1+1}, and indicates that there are $n - n_1$ free bytes of memory available. After two other arrays of size n_2 and n_3 are allocated, the free memory looks like:

$$M_{n_1+n_2+n_3+1}, \ldots, M_n,$$

and the free memory pointer list contains a single entry which points to memory element $M_{n_1+n_2+n_3+1}$, and indicates that there are $n - n_1 - n_2 - n_3$ free bytes of memory available. If the middle array (of size n_2) is de-allocated, the free memory will look like:

$$M_{n_1+1}, \ldots, M_{n_2}, M_{n_1+n_2+n_3+1}, \ldots, M_n.$$

Now there would be two elements in the free memory pointer list. The first element in the free list would point to memory element M_{n_1+1}, and indicate that n_2 free bytes of contiguous storage were available. The second element in the list would point to memory element $M_{n_1+n_2+n_3+1}$, and indicate that there are $n - n_1 - n_2 - n_3$ free bytes of memory available. Future memory allocations would now have to traverse the "free list" looking in each element to see if it contains enough contiguous storage to hold the amount of space required by the array. As such, memory allocation could no longer be considered an $O(1)$ operation.

Contiguous arrays are important and are necessary to justify the assumption of unit-time array lookup and update, both of which require calculating an offset before updating or reading. Because of these considerations, $dealloc_k$ is treated as a "no op" in our model. This helps to keep the model reasonably simple.

7.2.3 Costs of Reductions

Table 2 contains the costs associated with each reduction step. The notation $\mid f \mid$ refers to the cost of making a copy of the expression represented by f and is discussed in detail in the subsequent explanation of **repeat**. Suppose that $- \vdash_L^{\text{imp}} M\colon\theta$. If $M, s \rhd_E^* v, s'$ where v is a value (a natural number, location, λ-abstraction, pair, or $\otimes$-pair of values), the cost of evaluating M is the sum of the costs of each step in the reduction to v.

Before continuing, some of the rules and their associated costs are examined in detail, as a means of informally justifying that the costs are reasonable. The rationale behind the costs of array operations has already been discussed. However, it is worth repeating that the fact that update is being done destructively is crucial for the cost of one for this operation. Such a result would *not* be justified in a language without a coincidence result. The problem is that, without destructive overwrite, extra space would possibly have to be allocated to hold the result of an update.

Of the other rules, the crucial one is β-reduction (function application) which is assigned a cost of one. This is because β-reduction may be thought of in terms of

Reduction Rule			*Cost*
$(\lambda x.M)N, s$	$\triangleright_E$	$M[N/x], s$	1
$\mathbf{let}\ x \otimes y\ \mathbf{be}\ M \otimes N\ \mathbf{in}\ C, s$	$\triangleright_E$	$C[M/x, N/y], s$	2
$\pi_i\langle M_1, M_2\rangle, s$	$\triangleright_E$	M_i, s	1
$\mathbf{let}\ \star\ \mathbf{be}\ \star\ \mathbf{in}\ C, s$	$\triangleright_E$	C, s	1
$\mathit{ifz}(\underline{0} \otimes M), s$	$\triangleright_E$	$\pi_1 M, s$	1
$\mathit{ifz}(\underline{n} \otimes M), s$	$\triangleright_E$	$\pi_2 M, s$	1
$+(\underline{n} \otimes \underline{m}), s$	$\triangleright_E$	$\underline{n+m}, s$	1
$\mathbf{repeat}\ fl, s$	$\triangleright_E^{(k+1)}$	l', s' if $(fl), s \triangleright_E^k \underline{0} \otimes l', s'$	$1 + \mathit{cost}(f\ l)$
$\mathbf{repeat}\ fl, s$	$\triangleright_E^{(k+1)}$	$\mathbf{repeat}\ fl', s'$ if $fl, s \triangleright_E^k \underline{n} \otimes l', s'$	$1+ \mid f \mid + \mathit{cost}(fl)$
$\mathit{alloc}_k(\underline{n}), s$	$\triangleright_E$	$\ell, (s \mid \ell \mapsto \underline{n}, \cdots, \ell + (k-1) \mapsto \underline{n})$	k
$\mathit{update}_k(\underline{n}, \ell, i), s$	$\triangleright_E$	$\ell, (s \mid (\ell + i) \mapsto \underline{n})$	1
$\mathit{lookup}_k(\ell, i), s$	$\triangleright_E$	$(\ell, s(\ell + i)), s$	1
$\mathit{dealloc}_k(\ell), s$	$\triangleright_E$	$\star, s - \{\ell, \cdots, \ell + (k-1)\}$	0
$\mathit{copy}(\underline{n}), s$	$\triangleright_E$	$\underline{n} \otimes \underline{n}, s$	2
$\mathit{discard}(\underline{n}), s$	$\triangleright_E$	$\star, s$	1

Table 2: Costs of Reduction Rules

closures. Instead of performing a substitution $M[N/x]$, a closure is formed in which x holds a pointer to N, or a closure for N. Passing a pointer is regarded as having cost 1. From an implementation standpoint, the address of N is being stored into x, and then the evaluation of M continues. Since a call-by-value evaluation strategy is being used, it is important to note that N will already have been evaluated before its address is copied into x. Since the operation amounts to a simple copying of a single address, we assign β-reduction unit cost. Section 7.2.4 discussed β-reduction and explains its rationale in greater detail.

It is worthwhile to note that in the language of Rudimentary Types, variables only occur uniquely within a term. (See Definition 2 and Proposition 3 in Chapter 2 for a detailed more detailed discussion of uniqueness.) The other rules are straightforward. The **let** rule involves two substitutions (as opposed to the single substitution required in β-reduction), so the cost charged to this reduction is 2. In the rule for addition, it is assumed that the numbers being added together are sufficiently small so as to only take $O(1)$ to accomplish. (See the previous subsection for the list of assumptions.)

The cost associated with reducing a **repeat** seems unusual. It is made up of three parts. The assignment of 1 is the cost of checking whether the returned value n is zero. The charge of $cost(f\ l)$ is simply the cost of applying the function to the argument to produce a pair, $l' \otimes n$. The explanation for charging a cost of $\mid f \mid$ is more difficult to explain. Another iteration of the **repeat** loop will be required if the value $\underline{n}$ is non-zero. However, because the function will have to be applied to the argument l' during the next iteration of the **repeat** loop, it must first be copied. The function needs to be used twice: in the evaluation of the $(f\ l)$ expression and also for the next iteration of **repeat**. Because the actual structure of the function (imagine that a function, like any other structure in the language, can be represented as a parse tree) is destroyed when applied to its argument, it is first necessary to copy the entire parse tree (which represents the function) before actually applying the function to its argument. A cost of $\mid f \mid$ is assigned, which represents the cost of actually making a physical copy of the parse tree that expresses the function, f. For the **repeat** case when applying f to l will result in a value of $\underline{0}$, the cost is just $1 + cost(f\ l)$. The function may be destroyed as it will not be needed for another iteration of the **repeat** loop (as there will not be one).

The function $alloc_k$ actually allocates (and initializes) k memory locations, so its cost is k. For both $lookup_k$ and $update_k$, it is assumed that the particular location within the array can be accessed in unit time (see the subsection below), so that there is only a charge of 1 for each of these operations. Since $dealloc_k$ is a "no op", 0 is charged for this operation. As *copy* makes two copies of a variable of type **nat**, a cost of 2 is charged.

The problem of producing accurate cost models for λ-calculus is difficult, and is an active area of interest. (See [ML96, ML97] and its references.) The decisions made in the cost model may seem rather arbitrary and ad hoc, and the justification of many of those decisions is presented only informally. Additionally, the model is rather crude. It would be useful to make the implicit closure model that was appealed to more explicit, and to have a model that was validated over a period of time by a working implementation.

This would be a big job indeed, going well beyond the scope of this thesis. So it must be left for future work. But while there is a certain discomfort with the model's justification, the reasons given do carry some weight. The most important point is that destructive update, as justified by the coincidence theorem, forms the underpinning for the cost of the array operations. This is the key reason why depth-first search exhibits an an $O(v + e)$ running time under our cost model.

The next section illustrates the resulting assembler instructions a compiler for this language might generate. The purpose of this section is to argue informally for the validity of the cost model.

7.2.4 Implementation Sketch

Here, the approximate code is given that would be generated as a result of some of the reduction rules. The assembler instructions used are as specified in [Pap94].

The implementation (generation of the pseudo-assembler code) is accomplished by means of a stack. Informally, the instructions, together with their partial evaluations, are stored on the stack. The overall generation of the assembler code is accomplished by starting at the top of the stack and going down the stack until an expression needing evaluation (i.e., generation of assembler code) is reached. The evaluation occurs, the assembler code is generated, and the stack may be modified. We illustrate this with an example. Consider:

let	$@x$	$@y$	v_1	v_2	$@C$

Notationally, this expression indicates that, at the top of the stack, there is a pointer to the word **let**, indicating that the next five fields will be used in its evaluation expression. Next down, there is a pointer to the variable x, followed by a pointer to the variable y. Next down, there is a value, v_1. (Recall the definitions of values, Val, from section 4.4.) Next down, there is a value v_2. Next down, there is a pointer to the variable y. Next down, there is a pointer to the variable x. Finally, below that, there is the expression C. So, the next assembler instructions to be generated are for the evaluation of the **let** expression. For the sake of argument, it is assumed that register 15 ($R15$) points to the correct location on the stack, indicating the expression to be evaluated. The assembler instructions that need to be generated for some of the more interesting cases are illustrated.

Case: The topmost statement needing evaluation is of the form: $+(\underline{m} \otimes \underline{n})$. Then, the top of the stack is of the form:

$+$	$\underline{m}$	$\underline{n}$

The following instructions perform the reduction:

$LOAD$	$(R15+4)$	loads m into R0 – assuming 4 byte fields
ADD	$(R15+8)$	adds n to the contents of R0
$STORE$	$(R15+8)$	stores result
$LOAD$	$\uparrow R15$	now update R15 to point
ADD	8	to the result
$STORE$	$R15$	

Case: The topmost statement needing evaluation is of the form: $(\lambda\ x.M)N$. Then, the top of the stack is of the form:

Apply	$@x$	$@M$	$@N$

The following instructions perform the reduction:

$LOAD$	$\uparrow(R15+12)$	Loads address (closure) of N into R0
$STORE$	$\uparrow(R15+4)$	sets x to point to this closure
$LOAD$	$\uparrow(R15+8)$	Loads the address of M
$STORE$	$\uparrow R15$	Updates R15 to point to the next expression to be evaluated

It is interesting to note the destructive update of x, the variable being abstracted. As noted previously, this is acceptable because of linearity, as the variable x occurs uniquely within M. Such a means of implementing application would not work for the ordinary λ-calculus. In the ordinary λ-calculus, it would be necessary to make a copy of the term $\lambda x.M$, as there could be many pointers to the term. Destructively updating the term could (potentially) impact other parts of the program that also point to the term. Because of uniqueness, it is possible to destructively overwrite x to point to the closure of N.

Case: The topmost statement needing evaluation is of the form: **let** $x \otimes y$ **be** $v_1 \otimes v_2$ **in** C. Then, the top of the stack is of the form:

let	$@x$	$@y$	v_1	v_2	$@C$

The following instructions perform the reduction:

$LOAD$	$\uparrow(R15+12)$	Loads address of v_1 into R0
$STORE$	$\uparrow(R15+4)$	sets x to point to this closure
$LOAD$	$\uparrow(R15+16)$	Loads address of v_2 into R0
$STORE$	$\uparrow(R15+8)$	sets y to point to this closure
$LOAD$	$\uparrow(R15+20)$	Loads the address of C
$STORE$	$\uparrow R15$	Updates R15 to point to the next expression to be evaluated

This reduction is very similar to the previous one, except that two substitutions are required instead of one.

Case: The topmost statement needing evaluation is of the form: $update_k(val, \ell, i)$. Then, the top of the stack is of the form:

$update$	val	$@\ell$	i	k

The following instructions perform the reduction:

$LOAD$	$(R15 + 16)$	First checks if index
SUB	$(R15 + 12)$	is in bounds
$JNEG$	$\Omega_{\mathbf{cell}}$	
$JZERO$	$\Omega_{\mathbf{cell}}$	
$LOAD$	$(R15 + 8)$	Loads the array's address
ADD	$(R15 + 12)$	Adds the offset
$STORE$	$R1$	
$LOAD$	$(R15 + 4)$	
$STORE$	$\uparrow R1$	destructively update the array
$LOAD$	$(R15 + 8)$	
$STORE$	$R15$	Return the pointer to the array

Again, it is interesting to note that this form of destructive update would not work for a standard functional language.

Case: The topmost statement needing evaluation is of the form: **repeat** f l. Then, the top of the stack is of the form:

repeat	f	$@\ell$	$extra$

As **repeat** is more complicated than the rest of the instructions, a "pseudo-assembler code" series of instructions is presented:

newlabel:

Copy f to f'.

Store the address of f' in the unused space.
Apply f to ℓ, obtaining $n \otimes \ell'$.
Move the address of f' into $(R15 + 4)$.
Move the address of ℓ' into $(R15 + 8)$.
Test if $n = 0$.
If it is not, go to newlabel.
Store @ℓ' in R15.

It is necessary to make a copy of f before destructive applying ℓ to it, in case it is necessary to go through the loop again. The actual cost involved in the evaluation of the **repeat** statement may be slightly different from what was charged in the cost model, for the case that $(f\ l)$ results in a value $0 \otimes l'$. This is because, at the time the copy of f is made, it is not possible to know whether or not it will be necessary to go through the loop again. As such, it is necessary to make a copy of the function in case the resulting value of n is non-zero. Therefore, a cost of $1+ \mid f \mid +cost(f\ l)$ must be charged whenever a **repeat** statement is evaluated, regardless of the value of n that results.

7.3 A DFS algorithm

This section presents our version of Tarjan's depth first search algorithm, written in the language of Rudimentary Types. Initially, a very brief overview of the algorithm is provided. A comprehensive description of the algorithm, together with its proof of correctness and run time, can be found in [HT73] and [Tar72]. A more general discussion of depth-first search appears in [CLR90] and [AHU83]. The actual algorithm, written in the language of Rudimentary Types, is presented in the next part of this section. For comparison, the same algorithm written in Fortran77 (which is much easier to read) is provided. The last part of this section involves an informal argument for the algorithm's runtime, based on the complexity model provided earlier. It is important to note that a *single* DFS algorithm is not provided. Rather, there are a family of DFS algorithms, one for each size (i.e., number of nodes) of graph. (Pointerless Pascal and Fortran77 without dynamic arrays would also need to provide such a family of solutions.)

7.3.1 The algorithm

Depth First Search is an algorithm for visiting all of the nodes in a graph. It is inherently imperative, as its success of running in $O(v + e)$ (where v is the number of vertices and e is the number of edges) depends upon the ability to destructively update an array which is used to indicate which nodes have been previously visited.

Normally, the algorithm takes some sort of graph as input. However, since the language of Rudimentary Types does not contain pointers, the input is instead an $N \times N$ adjacency matrix (where each row corresponds to a vertex, and each column within that row corresponds to a node which is directly connected to the node in question). It is assumed that there is never more than one edge connecting two vertices. Because the language of Rudimentary Types does not have variable arrays, it is necessary to define a family of depth first search algorithms, dfs_N, one for possible number N of nodes. Here is the algorithm, written in the language of Rudimentary Types. There is a fair bit of syntactic sugar added to improve readability — in particular, $(\lambda x.M)N$ is occasionally replaced by **let** x **be** N **in** M.

$$
\begin{array}{lll}
dfs_N & \equiv_{def} & \lambda map.\ \ \mathbf{let}\ color\ \mathbf{be}\ alloc_N(0)\ \mathbf{in} \\
& & \quad \mathbf{let}\ loc\ \mathbf{be}\ alloc_N(0)\ \mathbf{in} \\
& & \quad\ \ \mathbf{let}\ whichvertex\ \mathbf{be}\ alloc_N(0)\ \mathbf{in} \\
& & \quad\ \ \ \ \mathbf{let}\ i\ \mathbf{be}\ alloc_1(1)\ \mathbf{in} \\
& & \quad\ \ \ \ \ \ \mathbf{let}\ i' \otimes color' \otimes loc' \otimes whichvertex' \otimes map'\ \mathbf{be} \\
& & \quad\ \ \ \ \ \ \ \ \mathbf{repeat}\ alg(i\ color\ loc\ whichvertex\ map)\ \mathbf{in} \\
& & \quad\ \ \ \ \ \ \ \ \mathbf{let}\ \star\ \mathbf{be}\ dealloc_N(color')\ \mathbf{in} \\
& & \quad\ \ \ \ \ \ \ \ \ \ \mathbf{let}\ \star\ \mathbf{be}\ dealloc_N(loc')\ \mathbf{in} \\
& & \quad\ \ \ \ \ \ \ \ \ \ \ \ \mathbf{let}\ \star\ \mathbf{be}\ dealloc_N(whichvertex')\ \mathbf{in} \\
& & \quad\ \ \ \ \ \ \ \ \ \ \ \ \ \ \mathbf{let}\ \star\ \mathbf{be}\ dealloc_N(i')\ \mathbf{in} \\
& & \quad\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ map'
\end{array}
$$

This routine allocates three arrays, *loc*, *whichvertex*, and *color* that will be used by the actual algorithm doing the depth first search. The *color* array is the essential imperative feature of the DFS algorithm. It is updated whenever a vertex is visited, so that it will never erroneously be visited at a later stage. Because there is no recursion, the algorithm needs two parallel arrays to keep track of its local "state". For example, if the algorithm has reached node m, and one of the nodes in m's adjacency list is node n, then before going on to do a DFS on node n, the algorithm must first stack where in node m's adjacency list it is (as well as the fact that it is presently doing a DFS on node m), so that when it finishes doing a DFS on node n, it will know where to return to, in node m's adjacency list. The single element array i is used by the algorithm to try to do a DFS starting from each node.

The main part of the algorithm, *alg*, is given below. As some of the expressions that make up this function are quite long, they are separated out of the main part of the algorithm for readability.

$$
\begin{array}{lll}
alg & \equiv_{def} & \lambda i.\lambda color.\lambda loc.\lambda whichvertex.\lambda map. \\
& & \mathbf{let}\ i' \otimes ival\ \mathbf{be}\ lookup_1(i)\ \mathbf{in} \\
& & \quad ifz(\dot{-}\ (i \otimes +(N \otimes 1)), \langle 0 \otimes i' \otimes color \otimes loc \otimes whichvertex \otimes map, part1\rangle)
\end{array}
$$

The function first check if it has tried a DFS for all of the vertices. If it has, it returns. Otherwise, it goes on to evaluate the expression in $part1$.

$$\begin{array}{lll} part1 & \equiv_{def} & \textbf{let } color' \otimes colorval \textbf{ be } lookup_N(color, ival) \textbf{ in} \\ & & \quad ifz(colorval, \langle part2, \\ & & \qquad 1 \otimes update_1(+(ival,1), i', 1) \otimes color' \otimes loc \otimes whichvertex \otimes map \rangle) \end{array}$$

The function then checks to see if the node has been previously visited. If it has, it simply will update i and try the next vertex. Otherwise, it goes on to actually process the node, as defined in $part2$.

$$\begin{array}{lll} part2 & \equiv_{def} & \textbf{let } color' \textbf{ be } update_N(1, color, ival) \textbf{ in} \\ & & \quad \textbf{let } pos \textbf{ be } alloc_1(1) \textbf{ in} \\ & & \quad\; \textbf{let } loc' \textbf{ be } update_N(1, loc, 1) \textbf{ in} \\ & & \quad\;\; \textbf{let } whichvertex' \textbf{ be } update_N(ival, whichvertex, 1) \textbf{ in} \\ & & \quad\;\;\; \textbf{let } pos' \otimes whichvertex'' \otimes loc'' \otimes map' \otimes color'' \textbf{ be} \\ & & \quad\;\;\;\; \textbf{repeat } f(pos, whichvertex, loc', map, color') \textbf{ in} \\ & & \quad\;\;\;\; \textbf{let } \star \textbf{ be } dealloc_1(pos) \textbf{ in} \\ & & \quad\;\;\;\;\; 1 \otimes i'' \otimes color'' \otimes loc'' \otimes whichvertex'' \otimes map' \end{array}$$

The code that makes up $part2$ marks the current node as visited, and sets up the parallel arrays by putting the current node along with a position of 1 into the first slot into the "stack" (parallel arrays $whichvertex$ and loc). It then sets up a single element array which keeps track of where in the stack we are currently processing, and calls function f repeatedly until it finishes processing all of the nodes in its adjacency list.

$$\begin{array}{lll} f & \equiv_{def} & \lambda pos.\lambda whichvertex.\lambda loc.\lambda map.\lambda color. \\ & & \quad \textbf{let } pos' \otimes posval \textbf{ be } lookup_1(pos) \textbf{ in} \\ & & \qquad ifz(posval, \langle 0 \otimes pos' \otimes whichvertex \otimes loc \otimes map \otimes color, part3 \rangle) \end{array}$$

When the variable pos is decremented back down to 0, the DFS from the given node has been completed. Otherwise, it is necessary to go through the adjacency list of the node located in the *pos* row of the parallel arrays.

$$\begin{aligned} part3 \quad \equiv_{def} \quad & \textbf{let } whichvertex' \otimes k \textbf{ be } lookup_N(whichvertex, posval) \textbf{ in} \\ & \textbf{let } loc' \otimes j \textbf{ be } lookup_N(loc, posval) \textbf{ in} \\ & \textbf{let } map' \otimes color' \otimes j' \otimes k' \textbf{ be} \\ & \quad \textbf{repeat } find(map, color, j, k) \textbf{ in} \\ & \quad ifz(\dot{-}\ (j \otimes +(N \otimes 1)), \\ & \quad \langle 1 \otimes update_1(\dot{-}\ (posval \otimes 1), pos', 1) \otimes whichvertex' \\ & \qquad \otimes loc' \otimes map' \otimes color', part4\rangle) \end{aligned}$$

Here, the location from which the algorithm last left off (in the DFS) is found, and the algorithm tries to find out if there is another node of the adjacency list of for the current node (by repeatedly calling function $find$). If there is not, the algorithm "pops" this node off of the stack (by decrementing the value of *pos*), and tries this node's parent. If there is one, the algorithm goes on to the code in $part4$.

$$\begin{aligned} part4 \quad \equiv_{def} \quad & \textbf{let } loc'' \textbf{ be } update_N(j', loc', posval) \textbf{ in} \\ & \textbf{let } pos'' \textbf{ be } update_1(+(posval\ 1)\ pos'\ 1) \textbf{ in} \\ & \textbf{let } loc''' \textbf{ be } update_N(1, loc'', +(posval\ 1)) \textbf{ in} \\ & \textbf{let } map'' \otimes mapval \textbf{ be } lookup_{N,N}(map', k', j') \textbf{ in} \\ & \textbf{let } whichvertex'' \textbf{ be } update_N(mapval, whichvertex', +(posval\ 1)) \textbf{ in} \\ & \textbf{let } color'' \textbf{ be } update_N(1, color', mapval) \textbf{ in} \\ & 1 \otimes pos'' \otimes whichvertex'' \otimes loc''' \otimes map'' \otimes color'' \end{aligned}$$

This code updates the "stack" with the previous position, adds the new node onto the "stack", marks the node as visited, and goes on to look at that node's adjacency matrix.

$$\begin{aligned} find \quad \equiv_{def} \quad & \lambda map.\lambda color.\lambda j.\lambda k. \\ & \quad ifz(\dot{-}\ (j \otimes +(N \otimes 1)), \langle 0 \otimes map \otimes color \otimes j \otimes k, part5\rangle) \end{aligned}$$

In searching for the next node in a node's adjacency list, the code first checks to see if the algorithm has already passed the last possible node in the list. If so, the depth-first search from that node is done; otherwise, the algorithm goes on to the code in *part5*.

$$\begin{aligned} part5 \;\; \equiv_{def} \;\; & \textbf{let } map' \otimes mapval \textbf{ be } lookup_{N,N}(map, k, j) \textbf{ in} \\ & \quad ifz(\dot{-}\,(mapval \otimes 999), \langle 0 \otimes map' \otimes color \otimes j \otimes k, part6 \rangle) \end{aligned}$$

In searching for the next node in a node's adjacency list, the code next checks to see if the value in the adjacency list is some special value, say 999, indicating that there are no more adjacent nodes to the current node. If this is the case, the depth-first search starting from that node is done; otherwise, the algorithm goes on to the code in *part6*.

$$\begin{aligned} part6 \;\; \equiv_{def} \;\; & \textbf{let } color' \otimes colorval \textbf{ be } lookup_{N}(color, mapval) \textbf{ in} \\ & ifz(colorval, \\ & \;\; \langle 0 \otimes map' \otimes color' \otimes +(j\ 1) \otimes k, 1 \otimes map' \otimes color' \otimes +(j\ 1) \otimes k, \rangle) \end{aligned}$$

The code now check to see if the next node in the adjacency list has been visited. If it has, the recursion will continue; otherwise, it has found a new node to visit, and the code will return.

Even with the added syntactic sugar, this routine is still fairly complex as written in the language of Rudimentary Types. For reference, Figure 15 contains a Fortran77 subroutine for doing depth first search, coded in an analogous manner to the algorithm written in the language of Rudimentary Types.

7.3.2 Analysis

A detailed analysis of the depth first search algorithm can be found in [HT73] and [Tar72]. Here, there is only a discussion of the specific algorithmic and language design issues that impact the overall algorithm.

Cost of stack manipulation – *loc* and *whichvertex* arrays

A vertex may only be put onto the stack if it has not yet been visited (its value in the *color* array is 0). As soon as a node is placed onto the *whichvertex* array, its value

```
      SUBROUTINE DFS (Map, N)
      Integer N
      Integer Map(N,N), Color(N), Which(N), Loc(N), ii, j, k, pos

      DO 100 ii = 1,N
           color(ii) = 0
100   CONTINUE

      DO 200 ii = 1,N
      IF (color(ii) .EQ. 0) THEN
           color(ii) = 1
           pos = 1
C       loc refers to the location in the ii'th row of the
C       adjacency matrix to look for unvisited children nodes
           loc(1) = 1
           which(1) = ii
           DO WHILE (pos .GE. 1)
C         I will loop until I run out of nodes to visit, starting
C         with the newly visited node from the overall traversal
            k = which(pos)
            j = loc(pos)
            DO WHILE ((j.LE.N) .AND. (Map(k,j) .NE. 999) .AND.
    *              (color(map(k,j)) .NE. 0))
               j = j + 1
            END DO
            IF ((j .GT. N) .OR. (Map(k,j) .EQ. 0)) THEN
C           There are no new nodes to visit off of the current node,
C           so go back to where I left off with my parent node
              pos = pos - 1
            ELSE
C           because I've found a new node that hasn't yet been
C           visited, I'll mark where I am with my current node and
C           then start visiting the new node
              loc(pos) = j
              pos = pos + 1
              loc(pos) = 1
              which(pos) = map(k,j)
              color(map(k,j)) = 1
            END IF
           END DO
      END IF
200   CONTINUE
      RETURN
      END
```

Figure 15: Fortran77 code for implementing depth first search

in the *color* array is changed to 1, so it may never be placed into this array again. So, the overall cost of moving vertices in and out of the array is $2v$, or $O(v)$.

For the *loc* array, there is the same cost as there is for the *whichvertex* array, together with updates of individual elements each time a new vertex is found. Yet, there can still be only v vertices found in total, so the total cost of all of the updates to this array is $2v + v$, or still $O(v)$.

Cost of color array – updates and lookups

The cost of updating the *color* array is simply $O(v)$, since each element in the array is changed from 0 to 1 exactly once. An element in the *color* array is looked up once for each edge in the graph, plus once more for each vertex (in the outermost loop). So, the total cost of lookups is $O(v + e)$, and the total cost of lookups and updates is still $O(v + e)$.

Cost of lookups into map array

Each edge in the map array is checked, plus, for each vertex, there is potentially one additional lookup in each row (which determines that here are no more edges associated with a particular vertex). So, the total cost of lookups into the map array is $O(v + e)$.

Cost of repeats

In the actual implementation of the call by value model using closures, it seems that much of the body of the function, f, in the **repeat** $f\ \ell$ evaluation could be shared rather than copied, as the only place copying is required is for the part of the function that actually involves computation with the argument. However, for this work, a full cost of executing a **repeat** is assumed, $1+ \mid f \mid$. The overall cost of all **repeat**s is $O(v + e)$. It is possible that each **repeat** could cost the length of the entire program, but this cost, while quite significant, and certainly disappointing, is independent of the numbers of vertices and edges, so it is just increasing the cost by a constant, albeit large, value. As such, the overall complexity of the algorithm remains $O(v+e)$.

Overall, none of the arrays used in the implementation worsen the overall running time for the algorithm of $O(v+e)$.

7.4 Comparisons of the Language of Rudimentary Types with Fortran77

The language of Rudimentary Types has features that enable the encoding of many Fortran77 programs. In fact the only features that Fortran77 has that are not directly encodable into the Language of Rudimentary Types are:

1. Aliasing,
2. Dynamic Arrays,
3. A GOTO statement, and
4. Global variables.

`F77--` may be defined to be Fortran77 with these features eliminated. We could encode `F77--` programs directly into the Language of Rudimentary Types. Due to the absence of syntactic sugar in the language of Rudimentary Types, it is not practical to provide the actual encoding. Yet, it is interesting to note that the Language of Rudimentary Types is almost as powerful as `F77`. It is also worthwhile to note the features available in the Language of Rudimentary Types which are not available in Fortran77:

1. Higher order procedures and
2. Heap storage.

7.5 What Isn't Done

This sections concludes with remarks about two important tasks which remain to be accomplished.

1. A call-by-value closure model was not actually implemented. The implementation of the Language of Rudimentary Types was made well before the issues of complexity were considered. As such, the implementation was made by appropriately extending the Bologna Optimal Higher-order Machine [AGN95], which is a call-by-name graph reduction model of the λ-calculus. Thus, our implementation is not useful for analyzing our depth first search algorithm on real data, as a means of informally showing the justification of our complexity costs. Had we built the complexity model first, we would have started from scratch and written a call-by-value closure implementation of our language.

2. A rigorous proof of adequacy between our actual implementation and our call-by-value operational semantics is required. Here, we have only worked informally. To be formal, we will need to define the idea of "used uniquely" — to prove formally the necessary properties of linearity.

Both of these omissions should be done as part of future work.

Chapter 8

Conclusion

This concluding chapter consists of a brief summary of some of the results found in this thesis, and other observations made as a result of this study. Then, some future possible directions are suggested.

8.1 Thesis Summary

We note what we consider to be the interesting findings of this work:

1. The language of Rudimentary Types makes a good starting point for our study of languages based on linear logic. Using Rudimentary Types, it was possible to generate a language that was a non-trivial subset of Fortran77, which allowed "serious" (if, albeit ugly) programs to be written. It also made a good starting point to study attempts to use linear logic as a means of introducing destructive update into functional languages, as the language was simple and elegant.

2. An imperative operational semantics has been formulated, as well as the notion of Coincidence which, along with operational extensionality, provides a framework for studying the sense in which languages are "simultaneously purely functional and imperative".

3. As a result of many of the subtleties found in our studies of languages based on linear logic, it was seen to be necessary to be mathematically rigorous, especially in regards to proofs of adequacy.

4. It was shown that the intuition about "used once" was not necessary to obtain an imperative understanding of the language of Rudimentary Types. (The **repeat** combinator broke the intuitive notion of functions of linear type using their arguments exactly once – however, **repeat** did *not* break the more important mathematical principle of coincidence.)

5. (Non-polymorphic) linear types do not seem to readily allow for object oriented programming. The "pseudo-global" state required within an object is not modeled well using simply linear types. This makes linear functional programming appear weak, from the point of view of imperative programming.

6. Existing linear functional languages do not seem to work well with unrestricted recursion. Attempts to admit even simple recursive functions, such as *quicksort*, into Steadfast Types Revisited has resulted in typing rules becoming horribly complex.

7. The importance for a functional language to have an associated cost model was argued. Claims about the performance of a language/algorithm need to be backed up by mathematical proofs, something a cost model provides. Complexity considerations give a good criteria for evaluating the imperative character of a functional language.

8. Many of the limitations with linear functional programming (such as the problems with recursion, and with objects and classes) did not seem to be previously observed in the literature.

8.2 Future Directions

Some work that should be done in the near future is discussed in section 7.5. There are also a few other extensions to this work that seem interesting to pursue.

1. Would it be possible to allow arrays to contain data other than integers? What if arrays could contain procedures?

2. Would it be possible semantically to add pointers to the language, if the data that was pointed to could not be modified?

3. Would it be possible to add "read-only" types to the language? Perhaps some sort of "read-only" linear type procedure would make generalized recursion much easier to handle.

Finally, O'Hearn and Reynolds [OR97] have done work with polymorphic linear λ-calculus. From a programming perspective, this allows for the inclusion of objects and classes into a functional language. However, an imperative interpretation of linear polymorphic types has not yet been made. Developing an appropriate imperative operational model would be a significant accomplishment, but it is not yet clear how this can be done.

8.3 Summary

At this point, we cannot say that there is a reasonably satisfactory linear functional language that is simultaneously imperative and purely functional, but we do feel that interesting progress has been made. Overall, the main contribution of this work is a clarification of the properties satisfied by, and limitations of, some functional languages based on Linear Logic. In this we have found the use of an imperative operational semantics, with an explicit store, as a simple but effective technology for investigating linear types. We feel, in particular, that complexity considerations provide useful and sharp criteria for languages that attempt to support imperative as well as functional readings.

Bibliography

[Abr93] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111(1–2):3–57, April 12 1993.

[AG97] A. W. Appel and M. Ginsburg. *Modern Compiler Implementation in C: Basic Techniques*. Cambridge University Press, New York, 1997.

[AGN95] A. Asperti, C. Giovannetti, and A. Naletto. The bologna optimal higher-order machine. Technical report, University of Bologna, March 1995. Report UBLCS-95-9.

[AHU83] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, Massachusetts, 1983.

[Bar96] A. Barber. Dual intuitionistic linear logic. Preliminary Report, September 1996.

[BBdPH92] P.N. Benton, G.M. Bierman, V.C.V. de Paiva, and J.M.E. Hyland. Term assignment for intuitionistic linear logic. Technical report, University of Cambridge, Computer Laboratory, August 1992. Report 262.

[Bie97] G. M. Bierman. Observations on a linear pcf. Preliminary report, January 1997.

[Bra96] T. Braüner. *An Axiomatic Approach to Adequacy*. Ph.D. thesis, University of Århus, 1996.

[BS93] E. Barendsen and S. Smetsers. Conventional and uniqueness typing in graph rewrite systems. Technical Report CSI-R9328, Computing Science Institute, University of Nijmegen, December 1993.

[CGR96] Jawahar Chirimar, Carl A. Gunter, and Jon Riecke. Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming*, 6(2):195–244, 1996.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, New York, 1990.

[Fel87] M. Felleisen. *The Calculi of Lambda-v-CS-Conversion*. Ph.D. thesis, Indiana University, 1987.

[GH90] J. Guzmán and P. Hudak. Single-threaded polymorphic lambda calculus. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 333–343, Philadelphia, PA, 1990. IEEE Computer Society Press, Los Alamitos, California.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, pages 1–102, 1987.

[Gir89] J.-Y. Girard. Towards a geometry of interaction. In J. W. Gray and A. Scedrov, editors, *Categories in Computer Science and Logic*, volume 92 of *Contemporary Mathematics*, pages 69–108. American Mathematical Society, 1989.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 1989.

[Gun92] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. The MIT Press, Cambridge, Mass., and London, England, 1992.

[Hol88] S. Holmström. Linear functional programming. Draft paper, Chalmers University of Technology, 1988.

[How89] D. Howe. Equality in lazy computation systems. In *Proc. 4th Ann. Symposium on Logic in Computer Science*, pages 198–203, Washington, August 1989. Comp. Soc. Press.

[HT73] J. E. Hopcroft and R. E. Tarjan. Algorithm447: Efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.

[Jon97] N. Jones. *Computability and Complexity From a Programming Perspective.* The MIT Press, Cambridge, Massachusetts, 1997.

[KL95] D. King and J. Launchbury. Structuring depth-first search algorithms in haskell. In *Conf. Record 22nd ACM Symp. on Principles of Programming Languages*, pages 344–354, San Francisco, California, 1995. ACM, New York.

[Laf88] Y. Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.

[LM92] P.D. Lincoln and J.C Mitchell. Operational aspects of linear lambda calculus. In *Proc. 7th IEEE Symp. on Logic in Computer Science*, pages 235–247, 1992.

[Mac94] Ian Mackie. Lilac : A functional programming language based on linear logic. *Journal of Functional Programming*, 4(4):395–433, oct 1994.

[Mil77] R. Milner. Fully abstract models of typed λ-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

[ML96] H. Mairson and J. L. Lawall. Optimality and inefficiency: What isn't a cost model of the lambda calculus. In *1996 ACM International Conference on Functional Programming*, pages 92–101, 1996.

[ML97] H. Mairson and J. L. Lawall. On the global dynamics of optimal graph reduction. In *1997 ACM International Conference on Functional Programming*, 1997.

[O'H95] P. W. O'Hearn. Note on Algol and conservatively extending functional programming. *Journal of Functional Programming*, 6(1), 1995. Also in [OT97a], pages 89–100.

[OR97] P.W. O'Hearn and J.C. Reynolds. From Algol to polymorphic linear lambda-calculus. manuscript, 1997.

[OT] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. Accepted (Jan. 1995) for publication in the *J. ACM*. Preliminary version appeared as [OT93].

[OT93] P. W. O'Hearn and R. D. Tennent. Relational parametricity and local variables. In POPL [POP93], pages 171–184.

[OT97a] P.W. O'Hearn and R.D. Tennent, editors. *Algol-like Languages*, volume 1. Birkhauser, Boston, 1997.

[OT97b] P.W. O'Hearn and R.D. Tennent, editors. *Algol-like Languages*, volume 2. Birkhauser, Boston, 1997.

[Pap94] C. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, Massachusetts, 1994.

[Pit94] A. M. Pitts. Some notes on inductive and co-inductive techniques in the semantics of functional languages. Technical Report BRICS-NS-94-5, BRICS, Department of Computer Science, University of Århus, December 1994.

[Pit95] A. M. Pitts. Operationally-based theories of program equivalence. Lecture Notes from course given at Isaac Newton Institute for Mathematical Sciences, Cambridge, England, November 1995.

[Pit96] A.M. Pitts. Reasoning about local variables with operationally-based logical relations. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science*, pages 152–163, New Jersey, USA, 1996. IEEE Computer Society Press, Los Alamitos, California. Also in [OT97b].

[Plo75] G. D. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Plo78] G. D. Plotkin. *The category of complete partial orders: a tool for making meanings*. Lecture notes for the Summer School on Foundations of Artificial Intelingence and Computer Science, Pisa, June 1978.

[POP93] *Conf. Record 20th ACM Symp. on Principles of Programming Languages*, Charleston, South Carolina, 1993. ACM, New York.

[Rey78] J. C. Reynolds. Syntactic control of interference. In *Conf. Record 5th ACM Symp. on Principles of Programming Languages*, pages 39–46, Tucson, Arizona, 1978. ACM, New York.

[Rey81] J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372, Amsterdam, October 1981. North-Holland, Amsterdam. Also in [OT97a], pages 67–88.

[Sco72] D. S. Scott. Mathematical concepts in programming language semantics. In *Proc. 1972 Spring Joint Computer Conference*, pages 225–34. AFIPS Press, Montvale, N.J., 1972.

[Tar72] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[Wad90] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC-2 Working Conference on Programming Concepts and*

Methods, pages 347–359, Sea of Galilee, Israel, April 1990. North Holland, Amsterdam.

[Wad91] P. Wadler. Is there a use for linear logic? In *ACM/IFIP Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 1991. Proceedings of the 1991 Conference.

[Wad92] P. Wadler. The essence of functional programming. In *Conf. Record 19th ACM Symp. on Principles of Programming Languages*, Albuquerque, New Mexico, 1992. ACM, New York.

[Wad93] P. Wadler. A taste of linear logic. In *Mathematical Foundations of Computing Science (LNCS 711)*. Springer Verlag, 1993. Gdansk, Poland.

[WF93] S. Weeks and M. Felleisen. On the orthogonality of assignments and procedures in Algol. In POPL [POP93], pages 57–70. Also in [OT97a], pages 101–124.

Biographical Data

Name:	Stephen Cooper
Date and Place of Birth:	September 11, 1966 Washington, D. C.
Elementary & High School:	Charles E. Smith Jewish Day School, Rockville, MD. Graduated 1984
College:	Cornell University Ithaca, NY. BA, Chemistry and Math, 1988
Graduate Work:	Syracuse University Syracuse, New York Graduate Assistant: September, 1992 - May, 1996