Adding Imperative Programming to The Pattern Calculus

by

Quy Tuan Nguyen

a thesis submitted for the degree

Doctor of Philosophy

2005

Certificate of Authorship/Originality

I certify that the work in this thesis has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree except as fully acknowledged within the text. I also certify that the thesis has been written by me. Any help that I have received in my research work and the preparation of the thesis itself has been acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

Signature of Candidate

Production Note:
Signature removed prior to publication.

Acknowledgment

First and foremost, I would like to thank my supervisor Barry Jay whose help, stimulating suggestions and encouragement helped me in all the time of research for and writing of this thesis. Thanks to Murray Cole, Mohammad Hamdan, Ryan Heise, Gabi Keller, Hai Yan Lu, Clara Murdaca and David Skillicorn for valuable advice and suggestions.

Also, I would like to give my thanks to my family and friends for all support and encouragement.

Contents

List of Figures

Abstract

By focusing on data and flow control, imperative languages provide a finely grained and efficient mechanism for directly manipulating state and memory. By focusing on functions, polymorphism increases the modularity and reusability of programs. The pattern calculus gives a new account of polymorphism over arbitrary datatypes which has been used as the foundation for building the functional language FISh2. The power of the new polymorphism is not limited to a functional setting and it can be extended into an imperative setting. The main contribution of this thesis is to expand the pattern calculus with imperative features and implement this within a version of FISh2.

Two approaches are developed in expanding the calculus to imperative programming based on two setting: functional and imperative. Based on a functional setting, updatable locations are given separate location types; while based on an imperative setting, locations and their values share the same types. In both approaches, structured locations can be defined in the same way the calculus defines structured data. Hence, generic functions on locations can be defined by pattern-matching on (location) constructors. In that way, the power of the combination exceeds that of the boundary of functional or imperative alone. In particular, with the generic assignment function, we have a new approach on memory management which performs inplace update whenever it is reasonable to do so.

Similar ideas could be used to extend the power of parametric polymorphism to parallel programming. To illustrate the approach, a key problem is addressed in detail, namely, distributing a data structure over a network of processors.

Chapter 1

Introduction

In typed programming, polymorphism allows the same code to be used with different types, which results in more general and abstract implementations of programs. Polymorphism makes languages more expressive by increasing the reusability of programs, while still maintaining type safety.

The pattern calculus gives a powerful new approach to polymorphism. One of the key ideas in building polymorphic functions is pattern-matching programs in which each pattern might have a different type. With these pattern-matching, functions such as the generic plus and generic equality can work on arbitrary data. The pattern calculus also gives new account for representing data, all data structures can be constructed using names and a finite set of constructors. The theory of data representation gives a new form of polymorphism that enables to write a generic mapping function. The pattern calculus has been used as the foundation for the functional programming language FISh2 [27]. The expressive power of the calculus can be extended to other styles of programming. The thesis discusses expansions of the calculus to handle imperative programming and their implementation as expansions of FISh2.

Considering different coding styles and the trade-off of different properties, we provide two different approaches of combining the pattern calculus with imperative features. In the first approach, updatable locations are given separate location types while in the second, locations and their values share the same type. In both approaches, the use of assignment as a fully generic function provides a new approach to memory management where the inplace update is performed whenever it is reasonable to do so. More generally, one is able to combine polymorphism with fine control of memory to build simple, reusable and memory efficient programs.

A new way of memory management can also be exploited in parallel programming. With the existing tools in the pattern calculus, we can build a generic data distribution function. The distribution function can be seen as one step towards the expansion of the pattern calculus to parallel programming which supports the construction of generic parallel programs such as a generic parallel mapping operation.

Generic parallel programming was the main focus of the original thesis plan. However, during the research, the adding of imperative features turned out to be more significant and time consuming than estimated. As a result, the focus of the thesis changed to imperative programming.

1.1 Outline of the Thesis

The first chapter is an introduction of this thesis. After the outline (in this section), there is a section about the outcomes, personal contribution and significance of this thesis. Next is an overview about implementation involved in this thesis. The last part of the chapter is the literature review.

The second chapter is about the pattern calculus as introduced by Jay [35]. We do not include all parts of Jay's paper in the chapter, for example, the theory of how to represent data structures is omitted. The main purpose of the chapter is to give the motivations and to fix the notations for this thesis.

Without restriction, polymorphism interacts with imperative features in an ill-typed way. The third chapter discusses a modification of the typing rules of the pattern calculus that enables the addition of imperative features. The main reason for being stand-alone is that the chapter sets up the typings for the next two chapters.

Chapter 4 is an expansion of the pattern calculus with imperative features based on a functional setting. Based on the pattern calculus as a purely functional system, location types and terms are added on top. Creations of locations are explicit. We also add location constructors so that pattern-matching can work on constructed locations. The chapter also includes some additional imperative features such that while-loops, vectors, and generic output to enhance the usability of the system. Some examples to show the expressive power of the system are provided.

Chapter 5 is another expansion of the pattern calculus with imperative features but it is imperative based. All data can be assigned; locations and their values share the same types. Apart from these differences, this expansion is similar to the one in Chapter 4.

Chapter 6 introduces a generic data distribution function which is a key step in generic parallel programming.

Chapter 7 concludes the thesis with some indication of related future work.

1.2 Contribution

The pattern calculus by itself can be seen as a foundation for a functional programming language. The overall contribution of this thesis is to expand the pattern calculus to imperative and parallel programming. The theory is grounded in the corresponding implementation of the programming language FISh2.

An early version of the system in Chapter 4 was published as a joint paper [37] with the title *The Polymorphic Imperative: a Generic Approach to Inplace Update.* This paper describes an extension of an earlier version of the pattern calculus with imperative features. The focus of the paper is on a generic assignment function that does inplace update whenever it is possible.

1.2.1 Personal Contribution

The results in Chapter 3 were produced by me. The system in Chapter 4 was a join work of Jay, Lu [1] and me. The system in Chapter 5 was developed

[1]C. Barry Jay and H.Y. Lu, University of Technology, Sydney, Australia

by me. The results in Chapter 6 were join work of Jay, Hamdan [2] and me.

The implementations of the adding imperative features as in Chapters 4 and 5 were done by me based on the foundation produced by Jay [27]. The generic distribution function in Chapter 6 was implemented by me.

1.2.2 Significance

The choice of styles in programming between functional and imperative can be seen as a trade-off between expressive power and the direct manipulation of memory and state. Based on the pattern calculus, the generic combination of imperative and functional has the strengths of both styles. Programs written in the system benefit from the elegance and compactness of higher-order functions and polymorphism in the functional style, as well as the direct manipulation of memory and state of programs in the imperative style.

The power of the combination exceeds that of functional or imperative alone. One of the highlights is the generic assignment function (in source code) that compares the shapes of the memory and the new value so that inplace update is done whenever it is reasonable to do so. From the functional programming viewpoint, the generic assignment function can be seen as an efficient and generic approach to memory management. From the imperative programming viewpoint, it can be seen as a new and clean approach to updating large data structures.

As distribution is a key step in parallel programming, the study of generic distribution provides a new understanding towards generic parallel programming. Our choice of distribution is about maximizing the locality and work

[2]Mohamamd M. Hamdan, Yarmouk University, Irbid, Jordan

balance for all processors.

1.3 Implementation

All the calculi in this thesis have been implemented. All examples presented in this thesis have been written, tested and they produced expected result. All of the codes are available on request.

This section is an overview about implementation activities involved in this thesis. All the implementation was done in the programming language FISh2 [27]. The programming language FISh2 (and its successor **bondi**) was built originally by Jay to implement and test various ideas and approaches in language design. The language has been changing and evolving all the time. We use Concurrent Versions System (CVS) for backing up and branching purposes.

There are two sorts of implementation I did for this thesis. The first one is about compiler implementation for FISh2. The compiler is written in the programming language OCAML [51]. A typical circle of compiler implementation work involves:

- getting a stable version of FISh2 and creating a new CVS branch;
- removing unwanted features if necessary;
- adding imperative features into the branch;
- testing and debugging the added features;
- merging all or part of the added imperative features with others' work.

The second one is about writing and testing code in FISh2. A part of the FISh2 coding is to implement imperative features such as a generic assignment and a generic output function. Another part of the FISh2 coding is to test the compiler and to find out new algorithm such as a generic distribution function.

1.4 Background

1.4.1 Imperative Programming

Imperative is a style of programming where programs are explicit sequences of commands or statements. Imperative programming gives the computer a list of instructions to execute in a particular order. Typically, imperative programs [55, 56] contain declarations of variables, and zero or more assignments of values to them. In the imperative style, intermediate values are usually stored explicitly in variables. Type restrictions may be used for each variable, the set of values that can be assigned to a variable is determined at its declaration. Statements allow assignments to be sequenced and iterated. As a result, the order of execution is significant in imperative programs.

The imperative style, evolving from machine languages, is usually considered to have more direct interaction with the hardware and hence is regarded as more efficient than other styles [55]. However, the direct interactions also make it harder to reason about programs [1].

1.4.2 Functional Programming

An overview

Declarative programming describes to the computer a set of constraints and might let the computer figure out how to satisfy them. The constraints are usually represented as relationships between variables in terms of functions or inference rules. Declarative programming contains two main branches: logic programming and functional programming.

Functional programming languages are typically based on the lambda-calculus [21]. In a purely functional setting, a program consists of a set of (possibly recursive) function definitions and an expression [3]. Expressions in purely functional languages are referential transparent i.e. the value of an expression is determined at point of declaration. Referential transparency makes programs in purely functional languages easier to reason about.

With the use of higher-order functions and parametric polymorphism [44], programs written in functional languages are generally compact and elegant. However, purely functional programming lacks the ability to directly control the memory and state of a machine which can lead to inefficient programs [55].

Hindley-Milner Type System

A large class of functional languages are built on the Hindley-Milner type system [44]. A *type* is one of the following: a *type variable* (meta-variable X); the *unit type* 1; a *product of two types*; a *coproduct of two types*; or a *function from one type to another type.* The type schemes (meta-variable τ)

are given by universal quantification ($\forall$) of types by type variables.

$$T ::= X \mid 1 \mid T * T \mid T + T \mid T \rightarrow T$$
$$\tau ::= T \mid \forall X.\tau$$

The use of type schemes allows schematic type variables, i.e. different uses of a type scheme may instantiate a type variable in different ways.

let-Polymorphism

In the Hindley-Milner calculus, a term (meta-variables s, t) is: a term variable (meta-variable x); or a constant; or a lambda abstraction ($\lambda x.t$); or a let expression (**let** $x = s$ **in** t) or application of two terms.

By giving type schemes for term variables, we have the polymorphic uses of term variables in the body of a let expression. For example, in the expression:

$$\textbf{let}\ \mathsf{id}\ = \lambda x.x\ \textbf{in}\ (\mathsf{id}\ 9, \mathsf{id}\ \mathsf{true})$$

the type variable for the identity function has the type scheme $\forall X.X \rightarrow X$. The universal quantifier $\forall X$ allows X to be instantiated to any type, of integers, booleans, etc. Hence id can be applied to both 9 and true. This is called parametric (sometimes **let** or Hindley-Milner) polymorphism.

1.4.3 Combining Functional and Imperative Features

Some work tries to combine some functional features and imperative features in one setting. As an attempt to create an efficient tool for scientific applications with some functional style behaviours, Single Assignment C [54] was developed as a C-based language that supports concurrency deducibility. Pizza [52] is a Java-based language that tries to capture parametric polymorphism. Motivated by Generic Java (GJ) [14], Java [16] also has parametric polymorphism. FISh [28, 31] [3] is a shape-based array programming language that has higher-order functions and parametric polymorphism. Haskell programming language [38] uses monads [67] for input/output; it has a restricted form of stateful computation that retains referential transparency. The ML [47] family is considered to be functional languages with some imperative features. Detailed discussions about them will be the next subsection.

1.4.4 Adding Location Types and Terms into a Functional Setting

Many different lines of work e.g. [46, 45, 63, 41, 69, 47] try to provide functional languages with some advantages of the imperative style; typically by adding constants for creating, updating and getting values of reference cells, with control structures such as sequential composition.

Let us explore this using an ML-like syntax [51, 63]. The expression $\mathsf{ref}\ v$ creates a reference cell of reference type holding an initial value v. The expression $c := v$ updates the reference cell c with the new value v. The expression $!c$ gets the value held in the reference cell c. The expression $u; v$ sequences u and v. The expression $[\,]$ is the empty list. The expression $::$ is the infix operator for creating a list from a head and a tail. hd is a function returning the head of a non-empty list. (u, v) is a pair of two components

[3]This is FISh, not to be confused with FISh2.

u and v; fst and snd are two functions for extracting the first and second components of a pair.

Referential transparency no longer holds for references. Consider the program:

$$\textbf{let } x = \mathsf{ref}\ 9 \textbf{ in while } !x > 0 \textbf{ do } x := !x - 1 \textbf{ done}.$$

It terminates but replacing all the occurrences of x by ref 9 the new program does not terminate.

In a purely functional setting, all variables can have polymorphic type scheme. A typical example is a polymorphic empty list:

$$\textbf{let } x = [\,] \textbf{ in } (\mathsf{true} :: x, 1 :: x).$$

The **let** expression binds x to a polymorphic empty list. The use of polymorphism enables the first part of the pair to have x as a list of booleans and the second part of the pair to have x as a list of integers.

Unfortunately, the combination of polymorphism and updatable locations (references) does not come for free. One might let some bad programs get through such as the following ill-typed line of code:

$$\textbf{let } x = \mathsf{ref}\ [\,] \textbf{ in } x := \mathsf{true} :: !x; (\mathsf{hd}\ !x) + 1.$$

The **let** expression binds x to a reference cell initialized to an empty list. If the polymorphism is unconstrained, then the first part of the sequential composition uses the cell as a reference to a list of booleans and the second part of the sequential composition uses the cell as a reference to a list of integers; which results in a run time type exception when a boolean is added to an integer.

A naive attempt to detect such type errors is to require that a term whose type contains a reference type is monomorphic. However, that is not enough as the following example shows:

$$\begin{array}{ll}\textbf{let}\ \mathsf{fref}\ x = & \textbf{let}\ r = \mathsf{ref}\ x\ \textbf{in}\ (\textbf{fun}\ u \rightarrow !r,\ \textbf{fun}\ u \rightarrow r := u)\ \textbf{in} \\ \textbf{let}\ t = \mathsf{fref}\ [\,]\ \textbf{in}\ (\mathsf{snd}\ t)\ [9];\ \mathsf{true} :: ((\mathsf{fst}\ t)\ \mathsf{true}) & \end{array}$$

The types of fref and t do not employ reference types. However, unrestricted polymorphism for the reference in the closure of fref creates a similar ill-typed problem as in the previous example. If the type scheme of t is $\forall X, Y.(Y \rightarrow \mathsf{list}\ X) * (\mathsf{list}\ X \rightarrow \mathsf{unit})$ then using X in the first part of the pair as boolean type and in the second as integer type results in a run time type exception.

Next, we discuss some approaches for having both polymorphism and updatable references.

1. The system developed by Wright [69] introduces *value polymorphism* using a separate class of expressions called values:

Expressions	$e ::= v \mid e_1\ e_2 \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \mid \mathsf{ref}\ e \mid !e \mid x := e$
Values	$v ::= x \mid \lambda x.e$

   The system introduces two inference rules for let expressions. For typing the expression **let** $x = s$ **in** t, the variable x has a polymorphic type scheme only when s is a value. As none of the values can be actual updatable references, all the references in the system are monomorphic.

2. The 1990 version of Standard ML [46] defines *non-expansive* expressions, but it allows some type variables in the *expansive* case to be generalized. By detecting the creation of mutable values by ref operator, Standard ML (1990 version) introduces a subclass of *imperative type variables*; and any type substituted for an imperative type variable must contain only imperative type variables. When the bound body of a **let** expression is expansive, then imperative type variables are not generalized.

   The 1997 revision of Standard ML [47] adopts value polymorphism.

3. SML New Jersey (1993) [63] associates each variable with an associated integer measuring the "degree of weakness" or "strength". This is the number of function applications to be applied before a reference is actually created. If the type variable is unconstrained then the integer is infinity. Each function application reduces the "strength" of variables. Only variables with positive strength can be generalized.

4. Leroy [41] introduces *closure typing* to keep track of references embedded in functions by defining *labels* for all the functions in the closure and *constraints* of these labels on type variables. All the type variables contain constraints in themselves so that it is always clear which variables can be generalized. The main aim of this work is to reduce the impact of an imperative type system on a purely functional one.

1.4.5 Generic Parallel Programs

A current trend in computing is that the size of the problems to be solved is increasing. However, the speed of a single processor is coming to its physical limits because of thermal generation/dissipation properties and electronic signal speeds [60], [12] etc. Parallel (high performance) computers provide a may of solving the problems [43].

Parallel programming is more difficult to understand and to implement than sequential programming because of the extra obligations to manage distribution of data, synchronization and communication between processors. A typical example is the quick-sort algorithm for arrays. Implemented in C, the program is about 30 lines of code. However, a public available parallel version of the same algorithm using C plus MPI [61] is about 1700 lines [49]. Moreover, when dealing with structured data, the parallel program becomes more complicated because one must either separate the structure from the data by flattening of the data structure or manipulate structured data directly. Because of the complexity of parallel programming, the properties of modularity and reusability in parallel programs become more important than in sequential programming [5]. When dealing directly with parallel programs on structured data [57], one can take advantage of generic programming to make parallel programming more reusable by writing generic parallel programs [17, 18, 4, 50], i.e. one parallel program working on different sorts of data structures.

Chapter 2

Review of the Pattern Calculus

This chapter reviews the pattern calculus as introduced by Jay [35] with some minor modifications. All the modifications have been discussed with Jay. We do not aim to repeat all of the Jay's paper here but just give the motivations and tools needed later in the thesis.

2.1 Introduction

There is a significant class of operations that are shared by all data structures. The aim of generic programming is to have the same operations that work on arbitrary data types in a safe way. The pattern calculus [35] is a novel contribution to generic programming with new account of pattern-matching and data structures. One of the key ideas is that these operations are built on pattern-matching programs in which each pattern might have a different type. For example, the generic plus operation can be defined in the pattern calculus as a pattern-matching of four cases. The first two cases are for addition of two integers or two real numbers. The third case is to deal with compound data structures. The last one is the default one. These four cases are to cover all data structures. Details of the operation will be presented later in the chapter. With pattern-matching such as one for the generic plus, we have a generic way to traverse all paths of an arbitrary data structure. The idea (path polymorphism) is noticed but not given a prominent role on Jay's paper [35]. However, a system that supports path polymorphism is the core path of this chapter; and will be exploited later in this thesis.

The pattern calculus also gives a new account datatypes. Using the combinatory type system, type variables can represent structure as well as data. This is the important in typing some generic functions such as mapping and folding in the pattern calculus. Moreover, the ability to represent arbitrary data structures using a fixed set of built-in constructors provides another possibility for complete case analysis for all data structures.

The system satisfies standard properties such as reduction is Church-Rosser, reduction preserves typing, typings of terms are stable under type substitution, type inference is correct.

2.2 Types

Using the functorial type system [36], a typical data type will be represented by the application of a type F to another type (or tuple of types) of data X. The idea is developed further in the combinatory type system which is given in Figure 2.1. It contains rules for *type contexts* (meta-variable Δ), *types* (meta-variables S and T) and *type schemes* (meta-variable τ). A type context is a sequence (maybe empty) of distinct type variables. The judgment $\Delta \vdash$ asserts that Δ is a *well-form type context.* In the original paper [35], Jay defines a type to be one of the following: a type variable (meta-variable X); a *type constant* (meta-variable C); an *application* of two types. The judgment $\Delta \vdash T$ asserts that T is a *well-form type* in type context Δ. Also in the original paper, Jay introduces a type constant Function, and Function $S\ T$ is the type of *functions* from S to T. In this research, it proves useful to distinguish function types from applications. We propose a minor change here, that is a new type form called function type, for any two types S and T. We define $S \to T$ is the function type from S to T instead of using the application of the constant Function. The key point of the change is that, there is no type which is both a function type and an application of types. The *raw type schemes* are types under quantification by type variables. The judgment $\Delta \vdash_s \tau$ asserts that τ is a *well-form raw type scheme* in type context Δ. The free and bound variables of a type or raw type scheme are defined in the usual way, as is α -conversion of bound variables. A *type scheme* is an equivalence class of raw type schemes under α -conversion of bound variables [47]. A type scheme is *closed* if it has no free variables.

There are some primitive built-in type constants such as: primint (primitive integer number type); primfloat (primitive real number type); primchar (primitive character type). Other type constants such as int (proper integer number type); float (proper float number type); char (proper character type); bool (boolean type); list are introduced through abstract datatype declarations. Details about declared types are in Section 7 of the pattern calculus paper [35] and they are similar to ones of ML.

Type contexts (Δ)
$$\frac{}{\vdash} \qquad \frac{}{\Delta, X \vdash}\; X \notin \Delta$$

Types (S, T)
$$\frac{}{\Delta \vdash X}\; X \in \Delta \qquad \frac{}{\Delta \vdash C}$$

$$\frac{\Delta \vdash S \quad \Delta \vdash T}{\Delta \vdash S\; T} \qquad \frac{\Delta \vdash S \quad \Delta \vdash T}{\Delta \vdash S \to T}$$

Type schemes (τ)
$$\frac{\Delta \vdash T}{\Delta \vdash_s T} \qquad \frac{\Delta, X \vdash_s \tau}{\Delta \vdash_s \forall X.\tau}$$

Figure 2.1: The Combinatory Type System

Type substitution, type unification and most general unifier are defined in the usual ways [47].

A type is *data-form* if it is a type constant or an application of two types. A key property is that no data-form type is also a function type. For each type T, the *arity* of T is a natural number defined by: if T is a type constant or a type variable or an application of two types, then the arity of T is zero; if T is a function type $T_1 \to T_2$, then the arity of T is one plus the artity of T_2. The *arity* of a type scheme $\forall X.T$ is the arity of T.

2.3 Terms

In conventional functional languages, "interesting" terms are all functions. The pattern calculus balances the importance of functions and data structures.

Constructors (meta-variable c) are used to build data structures in the pattern calculus. These include *primitive datum terms* (meta-variable d)

of primitive integers, primitive floating point reals or primitive characters. Other examples of built-in constructors are un; evr; ths; bind; rep; and exn. Other constructors are introduced through abstract datatype declaration such as true; false; cons; nil. Details about declared constructors are in Section 7 of the pattern calculus paper [35]. The *operators* are given by: *datum operators* (with the overloading meta-variable d); and the *primitive equality* primequal. The *term constants* (meta-variable b) are given by: constructors; and operators. Each constant comes equipped with a given closed type scheme. Primitive integers, primitive floating point real numbers and primitive characters are of types (trivial type schemes) primint, primfloat and primchar, respectively. An example of a datum operator is the primplusint of type primint $\rightarrow$ primint $\rightarrow$ primint for adding two primitive integers. The constant primequal is to check if the two terms are the same constructor; and its type scheme is $\forall X, Y.X \rightarrow Y \rightarrow \mathsf{bool}$. There is also a special constructor exn with the type scheme $\forall X.X$ for handling exceptions.

Patterns (meta-variable p) are given by: *term variables* (meta-variable x, y); constructors; or *applications* of one pattern to another. The judgment $\Delta; \Gamma \vdash_{\circ} p : T$ asserts that p is a pattern of type T in context $\Delta; \Gamma$. The applicative pattern $p\, p_1$ requires that the contexts of p and p_1 are independent from each other. Also, each term variable cannot appear more than once in a pattern.

Raw terms (meta-variables s, t) are given by: term variables, term constants, *applications of terms*, *extensions*, let-terms and recursions. The judgment $\Delta; \Gamma \vdash t : T$ asserts that t is a raw term of type T in context $\Delta; \Gamma$. Free, bound variables and term substitution are defined in the usual way.

Free variables (fv) of an extension is defined as

$$fv\ (\mathbf{at}\ p\ \mathbf{use}\ s\ \mathbf{else}\ t) = fv(t) \cup (fv(s) - fv(p)).$$

Terms are defined as equivalent classes of raw terms under α -conversion of bound variables.

One of the key novelties of the calculus is the powerful branching and binding construct (named extension):

$$\mathbf{at}\ p\ \mathbf{use}\ s\ \mathbf{else}\ t$$

where s and t are called the *default function* and *specialization* of the extension respectively.

The type derivation rules of the pattern calculus are given in Figure 2.2. The type derivation for let-expression **let** $x = s$ **in** t allows the variable x a polymorphic type scheme. This works well for a purely functional system.

The term $\mathbf{fix}(x, t)$ is a *polymorphic recursion* with the *fix-point of* t with respect to the recursion variable x. It is called polymorphic because a type scheme is given to x when deciding the type of t. Different uses of the recursion variable in defining a generic function may exploit different instantiations of its type scheme.

A term is called an *explicit function* if it is an extension or a partially-applied constant. It is clear that all explicit functions (apart from terms headed by exn) have function types.

Next, we define some syntactic sugar. Note that some core notations

Term contexts (Γ)

$$\frac{\Delta \vdash}{\Delta;\ \vdash} \qquad \frac{\Delta;\Gamma \vdash \quad \Delta \vdash_s \tau}{\Delta;\Gamma, x:\tau \vdash}\ x \notin \Gamma$$

Patterns (p)

$$\frac{}{X; x : X \vdash_\circ x : X} \qquad \frac{}{\Delta;\ \vdash_\circ c : T}\ c : \forall \Delta.T$$

$$\frac{\Delta;\Gamma \vdash_\circ p : T \quad \Delta_1;\Gamma_1 \vdash_\circ p_1 : T_1 \quad \Delta,\Delta_1,X;\Gamma,\Gamma_1 \vdash}{\Delta,\Delta_1; v\Gamma, v\Gamma_1 \vdash_\circ p\ p_1 : vX}\ v = \mathcal{U}(T_1 \to X, T)$$

Terms (s,t)

$$\frac{\Delta;\Gamma \vdash}{\Delta;\Gamma \vdash x : \sigma T}\ \begin{array}{l}\Gamma(x) = \forall \Delta_1.T\\ \sigma : \Delta_1 \to \Delta\end{array}$$

$$\frac{\Delta;\Gamma \vdash}{\Delta;\Gamma \vdash b : \sigma T}\ \begin{array}{l}b : \forall \Delta_1.T\\ \sigma : \Delta_1 \to \Delta\end{array}$$

$$\frac{\Delta;\Gamma \vdash s : T \to S \quad \Delta;\Gamma \vdash t : T}{\Delta;\Gamma \vdash s\ t : S}$$

$$\frac{\Delta;\Gamma \vdash t : T \to S\ \Delta_1;\Gamma_1 \vdash_\circ p : T_1\ \Delta,\Delta_1; v\Gamma, v\Gamma_1 \vdash s : vS}{\Delta;\Gamma \vdash \mathbf{at}\ p\ \mathbf{use}\ s\ \mathbf{else}\ t : T \to S}\ v = \mathcal{U}(T_1, T)$$

$$\frac{\Delta;\Gamma \vdash t : T \to S \quad \Delta_1;\Gamma_1 \vdash_\circ p : T_1 \quad \Delta,\Delta_1;\Gamma,\Gamma_1 \vdash}{\Delta;\Gamma \vdash \mathbf{at}\ p\ \mathbf{use}\ s\ \mathbf{else}\ t : T \to S}\ \mathcal{U}(T_1, T) \uparrow$$

$$\frac{\Delta,\Delta_1;\Gamma \vdash s : S \quad \Delta;\Gamma, x : \forall \Delta_1.S \vdash t : T}{\Delta;\Gamma \vdash \mathbf{let}\ x = s\ \mathbf{in}\ t : T}$$

$$\frac{\Delta,\Delta_1;\Gamma, x : \forall \Delta_1.T \vdash t : T}{\Delta,\Delta_1;\Gamma \vdash \mathbf{fix}\ (x,t) : T}$$

Figure 2.2: The Original Pattern Calculus

of many other programming systems such as $\lambda x.t$ and **match**... **with** are syntactic sugar here.

The syntax $\mid p \rightarrow t$ is syntactic sugar for the program fragment

$$\textbf{at } p \textbf{ use } t \textbf{ else } .$$

A sequence of such fragments is a pattern-match with the ultimate exn default. For example, $\mid\ x \rightarrow x$ is syntactic sugar for **at** x **use** x **else** exn. In these pattern-matches, the wild-card symbol _ represents a fresh variable in the pattern part (p). The syntax **match** s **with** t is syntactic sugar for $t\ s$, especially when t is a pattern-match. The usual lambda abstraction $\lambda x.t$ is syntactic sugar for **at** x **use** t **else** exn. We use **at** p_1 **and** p_2 **use** s **else** t as syntactic sugar for

$$\begin{array}{l}\textbf{at } p_1 \\ \textbf{use at } p_2 \textbf{ use } s \textbf{ else } \lambda y.t\ (p_1\ y) \\ \textbf{else } \lambda x.\lambda y.t\ (x\ y).\end{array}$$

This extension tries to match p_1 with x and p_2 with y with failure at any point applying the default to a reconstructed version of $x\ y$.

As the usual syntax for defining functions, **let** $x\ p = s$ is syntactic sugar for **let** $x =$ **at** p **use** s **else** exn (especially when p is a variable). For example, **let** $f\ x = x + 1$ is syntactic sugar for **let** $f =$ **at** x **use** $x + 1$ **else** exn.

To define a recursive term using the let-expression, we use **let rec** $x = s$ as syntactic sugar for **let** $x = \textbf{fix}(x, s)$.

2.4 Constructed Terms

The *arity* of a term constant b is defined to be positive infinity if b is the exn constructor, and otherwise to be the arity of its given type scheme.

A term is *headed by* a term constant b if it is b itself or an application $s\ t$ where s is headed by b. The list of *arguments* of a term t headed by a constant is: the empty list if t is just the constant; is the concatenation of the t_2 with the list of arguments of t_1 if t is the application of t_1 to t_2. A term headed by a constant b is called a *fully-applied* constant if the number of arguments of the term equals the arity of b; and it is called a *partially-applied* constant if the number of arguments of the term is strictly smaller than the arity of b.

A *constructed term* is a term headed by a constructor. The application of a constructed term to another term is called a *applicative* constructed term. A term t *cannot become a constructor* if t is an applicative term or an extension or a partially-applied operator. A term t *cannot become applicative* if t is an extension or a constructor or a partially-applied operator. A pattern p and a term t *cannot match* in exactly the following two cases: first, when p is a constructor c, and either t is a constructor other than c or t cannot become a constructor; second, when p is a applicative term and t cannot become applicative. Two terms *cannot compare* in exactly the following two cases: first, (at least) one of them cannot become a constructor; second, they are two distinct constructors.

$$\begin{array}{rcl}
(\textbf{at}\ x\ \textbf{use}\ s\ \textbf{else}\ t)\ t_1 & > & s[t_1/x] \\
(\textbf{at}\ c\ \textbf{use}\ s\ \textbf{else}\ t)\ c & > & s \\
(\textbf{at}\ p_1\ p_2\ \textbf{use}\ s\ \textbf{else}\ t)(t_1\ t_2) & > & (\textbf{at}\ p_1\ \textbf{and}\ p_2\ \textbf{use}\ s\ \textbf{else}\ t)\ t_1\ t_2 \\
 & & \quad \text{if } t_1 \text{ is a constructed term} \\
(\textbf{at}\ p\ \textbf{use}\ s\ \textbf{else}\ t)\ t_1 & > & t\ t_1 \quad \text{if } p \text{ and } t_1 \text{ cannot match} \\
\textbf{let}\ x = s\ \textbf{in}\ t & > & t[s/x] \\
\textbf{fix}\ (x,t) & > & t\{\textbf{fix}\ (x,t)/x\} \\
d_o\ d_1\ \ldots d_k & > & d' \quad \text{if } d_o\ d_1\ \ldots d_k \text{ equals } d' \\
\mathsf{primequal}\ c\ c & > & \mathsf{true} \\
\mathsf{primequal}\ t\ t_1 & > & \mathsf{false} \quad \text{if } t,\ t_1 \text{ cannot compare}
\end{array}$$

Figure 2.3: Reduction Rules

2.5 Reduction

The *basic reduction rules* (denoted by relation $>$) are given in Figure 2.3. We will discuss all the cases.

There are four reduction rules for the application of an extension to a term. If the pattern is a variable x then specialization occurs with t_1 substituting for x as the argument. If the pattern is a constructor c and the argument is also c then the result is the specialization s. If the pattern is an application $p_1\ p_2$ and the argument is an applicative term $t_1\ t_2$ then specialization occurs. The specialization tries to match p_1 with t_1 and p_2 with t_2 with failure at any point applying the default to a reconstructed version of $t_1\ t_2$ as defined in Section 2.3. If the pattern and the argument cannot match then the result is the application of the default to the argument. The reduction of let-terms substitutes the bound variable with the body of the let. The reduction of a fix-point results in its body with the recursion variable replaced by the fix-point. The reduction of a fully applied datum operator results in the expected datum. The reduction of the application of primequal to two terms results in: true if the two terms are the same constructor; and false otherwise.

We present here one lemma about the property of term substitution which will be used in later chapters of the thesis.

Lemma 1 *Typings of terms are stable under term substitution. That is, if there are derivations* $\Delta; \Gamma, x : \forall \Delta_1.S \vdash t : T$ *and* $\Delta, \Delta_1; \Gamma \vdash s : S$ *then there is a derivation of* $\Delta, \Delta_1; \Gamma \vdash t[s/x] : T$.

Proof: The proof is by induction on the structure of the derivation of the typing of t. We do a case analysis on the type rules in Figure 2.2. All the cases are standard.

variable If t is variable x then $t[s/x]$ is s. We know that $\Delta; \Gamma \vdash x : \sigma T$ where $\sigma : \Delta_1 \rightarrow \Delta$. Combining that with the assumption $\Delta; \Gamma, x : \forall \Delta_1.S \vdash x : T$ and $\Delta, \Delta_1; \Gamma \vdash s : S$. So we can conclude that $\Delta, \Delta_1; \Gamma \vdash t[s/x] : T$. If t is variable different from x then from the assumption $\Delta; \Gamma, x : \forall \Delta_1.S \vdash t : T$ we can deduce that $\Delta; \Gamma \vdash t : T$ which is the same as $\Delta, \Delta_1; \Gamma \vdash t[s/x] : T$.

constant If t is a constant b then $t[s/x]$ is just b. From the assumption $\Delta; \Gamma, x : \forall \Delta_1.S \vdash c : T$ we can deduce that $\Delta; \Gamma \vdash c : T$ which is the same as $\Delta; \Gamma \vdash c[s/x] : T$ (as $\Delta, \Delta_1; \Gamma \vdash$).

application If t is an application $t_1\ t_2$ then there is a type T_1 such that $\Delta; \Gamma, x : \forall \Delta_1.S \vdash t_1 : T_1 \rightarrow T$ and $\Delta; \Gamma, x : \forall \Delta_1.S \vdash t_2 : T_1$. Applying the induction hypothesis for t_1 and t_2 we know that $\Delta, \Delta_1; \Gamma \vdash$

$t_1[s/x] : T_1 \to T$ and $\Delta, \Delta_1; \Gamma \vdash t_2[s/x] : T_1$. Using the typing rule for application we can conclude that $\Delta, \Delta_1; \Gamma \vdash (t_1\ t_2)[s/x] : T$.

fix-point If t is of the form $\mathbf{fix}(x_1, t_1)$, with α -conversion we can assume that x_1 is not in $\Gamma, x : \forall\Delta_1.S$. The only way to get the type $\Delta; \Gamma, x : \forall\Delta_1.S \vdash \mathbf{fix}(x_1, t_1) : T$ is by $\Delta', \Delta''; \Gamma, x : \forall\Delta_1.S, x_1 : \forall\Delta''.T \vdash t_1 : T$ (where $\Delta = \Delta', \Delta''$). Applying the induction hypothesis for t_1 we have $\Delta, \Delta_1; \Gamma, x_1 : \forall\Delta''.T \vdash t_1[s/x] : T$. With the typing rule for fix-point, we can deduce that $\Delta, \Delta_1; \Gamma \vdash \mathbf{fix}(x_1, t_1[s/x]) : T$ and hence $\Delta, \Delta_1; \Gamma \vdash \mathbf{fix}(x_1, t_1)[s/x] : T$.

extension If t is of the form **at** p **use** t_1 **else** t_2, there are two sub-cases. The first sub-case is when x is not in the free variables of p. We know that $t[s/x]$ is **at** p **use** $t_1[s/x]$ **else** $t_2[s/x]$. Applying the induction hypothesis for t_1 and t_2 we have $\Delta, \Delta_1; \Gamma \vdash t[s/x] : T$. The second sub-case is when x is in the free variables of p. We know that x cannot be in the free variables of t_2 and hence x is not a free variable of t and the lemma holds for the sub-case.

let If t is of the form **let** $x_1 = t_1$ **in** t_2, there are two sub-cases. The first sub-case is when x is not x_1. We know that $t[s/x]$ is **let** $x_1 = t_1[s/x]$ **in** $t_2[s/x]$. Applying the induction hypothesis for t_1 and t_2 we have $\Delta, \Delta_1; \Gamma \vdash t[s/x] : T$. The second sub-case is when x is x_1. We know that x is not a free variable of t and the lemma holds for the sub-case.

$\square$

2.6 Examples

This subsection contains some standard utilities for a programming language. Most of them will be used in the later chapters of this thesis. Note that one can introduce new type constants and constructors using *simple type declaration* and *type declaration* in a similar way as in ML. For more details see Section 7 of the pattern calculus paper [35].

The pairing (binary product) is defined by:

$$\textbf{type}\ \mathsf{pair}\ X\ Y = \mathsf{pair}\ \textbf{of}\ X\ \textbf{and}\ Y.$$

We might use the usual syntax (x, y) for the term $\mathsf{pair}\ x\ y$ and $X * Y$ for the type $\mathsf{pair}\ X\ Y$. For example, $(9, \mathsf{true})$ is a pair (term) of the type $\mathsf{int} * \mathsf{bool}$.

The booleans are defined by

$$\textbf{type}\ \mathsf{bool} = \mathsf{true} \mid \mathsf{false}.$$

The infix-operation of conjunction (&&) and disjunction (||) and other operations on booleans such as $\mathsf{not} : \mathsf{bool} \rightarrow \mathsf{bool}$ can be defined in the usual way.

The standard conditional **if** b **then** s **else** t is syntactic sugar for the application $\mathsf{if\_then\_else}\ b\ s\ t$; where the function $\mathsf{if\_then\_else}$ is defined as

$$\begin{aligned}
&\mathsf{if\_then\_else} : \mathsf{bool} \rightarrow X \rightarrow X \rightarrow X = \\
&\mid \mathsf{true} \rightarrow (\ \mid s \rightarrow\ \mid \_ \rightarrow s) \\
&\mid \mathsf{false} \rightarrow (\ \mid \_ \rightarrow\ \mid t \rightarrow t).
\end{aligned}$$

Based on primequal the generic equality function equal can be defined as

```
equal : $X \to X \to$ bool =
let rec (equal0 : $X \to Y \to$ bool) =
| $x_0\ x_1 \to$ ( | $y_0\ y_1 \to$ if (equal0 $x_0\ y_0$) then (equal0 $x_1\ y_1$) else false)
| $x \to$ ( | $y \to$ primequal $x\ y$)
in equal0
```

Note that when the applications $x_0\ x_1$ and $y_0\ y_1$ share the same type, their components x_0 and y_0 might have different types. So, equal has to be defined through equal0 of a looser type $X \to Y \to$ bool. We face the same issue when defining some other generic binary functions such as generic addition and multiplication functions.

To pattern-match against integers as in the function plus bellow, the *proper integers* are abstract datatypes building on the primitive integers. The same method is used for floats and characters.

```
type int = int of primint
type float = float of primfloat
type char = char of primchar
```

In this thesis, primitive datum values are marked with enclosing $< \cdots >$. For example, the expression $<9>$ represents the primitive integer of value 9. The corresponding proper integer 9 is then int $<9>$. Character values are marked with enclosing $'\ldots'$ to avoid confusion with numbers (such as 9), normal spacing (blank character), or mathematical symbols. For example, proper character $'a'$ is then char $<'a'>$.

Similar to the generic equality function, the generic plus function can be defined using primitive operators primplusint and primplusfloat which add two primitive integers and two primitive floats respectively in:

```
plus : $X \to X \to X =$
**let rec** (plus0 : $X \to Y \to X$) $=$
| int $x_0 \to$ ( | int $y_0 \to$ int (primplusint $x_0$ $y_0$))
| float $x_0 \to$ ( | float $y_0 \to$ float (primplusfloat $x_0$ $y_0$))
| $x_0$ $x_1 \to$ ( | $y_0$ $y_1 \to$ (plus0 $x_0$ $y_0$) (plus0 $x_1$ $y_1$))
| $x \to$ ( | $y \to$ **if** primequal $x$ $y$ **then** $x$ **else** exn)
**in** plus0
```

We might use the usual infix-operation $+$ in the place of the generic plus function. For example $9+9$ is syntactic sugar for plus 9 9. Other generic arithmetic functions such as generic multiplication can be defined in a similar way to plus. As an example of specialized codes, we add an extra case for the multiplication of two complex numbers. Suppose complex numbers are defined as:

type Complex = Complex **of** primfloat **and** primfloat.

Base on primitive operators primmultint, primmultfloat and primminusfloat acting on primitive integers and primitive floats, the generic multiplication function can be defined as:

```
mult : $X \to X \to X =$
let rec (mult0 : $X \to Y \to X$) =
| Complex $x_0$ $x_1$ $\to$ ( | Complex $y_0$ $y_1$ $\to$ Complex
    (primminusfloat (primmultfloat $x_0$ $y_0$) (primmultfloat $x_1$ $y_1$))
    (primplusfloat (primmultfloat $x_0$ $y_1$) (primmultfloat $x_1$ $y_0$)))
| int $x_0$ $\to$ ( | int $y_0$ $\to$ int (primmultint $x_0$ $y_0$))
| float $x_0$ $\to$ ( | float $y_0$ $\to$ float (primmultfloat $x_0$ $y_0$))
| $x_0$ $x_1$ $\to$ ( | $y_0$ $y_1$ $\to$ (mult0 $x_0$ $y_0$) (mult0 $x_1$ $y_1$))
| $x$ $\to$ ( | $y$ $\to$ if primequal $x$ $y$ then $x$ else exn)
in mult0
```

The ability of defining patterns of different types and the applicative pattern (such as $x_0\ x_1$ in the generic equality) give a new form of polymorphism. Jay calls it *path polymorphism* [1]; which is a simple and generic way to traverse all paths of an arbitrary data structure.

The type of list is defined as:

type list X = nil | cons **of** X **and** list X.

The usual syntax for lists is accepted, such as [ ] for the empty list nil and $[1, 2, 3]$ for cons 1 (cons 2 (cons 3 nil)). The common operations of list: head

[1]The name was not in the original paper [35], but in later development of the pattern calculus, Jay gave it that name.

and tail are defined in the usual way:

$$\mathsf{tail} : \mathsf{list}\ X \rightarrow \mathsf{list}\ X = \ \mid \mathsf{cons}\ h\ t \rightarrow t$$

$$\mathsf{head} : \mathsf{list}\ X \rightarrow X = \ \mid \mathsf{cons}\ h\ t \rightarrow h$$

The type of string is defined as:

$$\textbf{type}\ \mathsf{string} = \mathsf{string}\ \textbf{of}\ \mathsf{list\ primchar}.$$

The usual syntax for strings is accepted, such as "*abc*" for the string

$$\mathsf{string\ (cons}\ <'a'>\ \mathsf{(cons}\ <'b'>\ \mathsf{(cons}\ <'c'>\ \mathsf{nil))).}$$

2.7 Representing Data Structures

This section is an overview about representation of data as discussed in Section 8 of the pattern calculus paper [35]. It contains some bases for constructing higher-order functions such as mapping or folding. However, the expansions with imperative features do not depend on the representation of data. In particular, after this chapter, we do not refer to this section.

A key step in understanding generic functions is to understand the nature of datatypes. Using category theory, there is a clear boundary separating structure from data [30]. Using the functorial type system [36], a typical data type will be represented by the application of a type F to another type (or tuple of types) of data X.

Generic functions in the pattern calculus such as mapping, folding and zipping employ parametric polymorphism in typing the structure part as well as in the data part. For example, one can generalize the function

$$\mathsf{maplist} : (X \rightarrow Y) \rightarrow \mathsf{list}\ X \rightarrow \mathsf{list}\ Y$$

to function map1 which works not only on list but any structure F:

$$\mathsf{map1} : (X \rightarrow Y) \rightarrow F\ X \rightarrow F\ Y$$

where X and Y are type variables, and F is a structure (type) variable. In the pattern calculus, map1 is a special case of a more general function map when there is only one sort of data.

As the generic mapping function can act on structures holding several sorts of data, we define a datatype arrow as a form of gathering functions from one tuple to another.

type arrow **has**
  $\mathsf{arrow}\ X\ Y = \mathsf{onefun}$ **of** $(X \rightarrow Y)$
or $\mathsf{arrow}(X_1, X_2)(Y_1, Y_2) = \mathsf{bthfun}$ **of** $\mathsf{arrow}(X_1, Y_1)$ **and** $\mathsf{arrow}(X_2, Y_2)$

Since all data structures can be represented using a fixed finite set of constructors [36], in order to write generic function working on arbitrary data type, we just need to give the cases for each constructor in the set. One example is the function

$$\mathsf{map} : \mathsf{arrow}\ X\ Y \rightarrow F\ X \rightarrow F\ Y.$$

Two of the common uses of mapping are map1 and map2 functions:

$$\mathsf{map1}\ f = \mathsf{map}\ (\mathsf{onefun}\ f)$$
$$\mathsf{map2}\ f\ g = \mathsf{map}\ (\mathsf{bthfun}\ (\mathsf{onefun}\ f)\ (\mathsf{onefun}\ g))$$

For example, map1 $(\lambda x.x + 1)$ acts on all data structures of integers (adding 1 to those). This is an example for another new form of polymorphism: *structure polymorphism.* [2]

2.8 Conclusion

The pattern calculus gives a new account of both pattern-matching and datatypes to provide new approaches for generic programming. With a new understanding about pattern-matching, one has the path polymorphism as a generic way to traverse arbitrary data structures. Details of path polymorphism are given in this chapter; as we will try to expand the power path polymorphism further in later chapter of the thesis. We also give an overview about structure polymorphism which is interesting but we are not going to exploit that later in the thesis.

When I was writing this thesis, the pattern calculus (Jay's research in generic programming) has been developed to support further new forms of polymorphism. However, due to the time limit, I could not work much on the newer version of the pattern calculus; this thesis focuses on path polymorphism as in [35].

[2]Again, the name was not in the original paper, but appeared in later development of the pattern calculus.

Chapter 3

Restricting Polymorphism to Functions

This chapter introduces some modifications of the pattern calculus that will better support the imperative features considered in the next two chapters.

3.1 Introduction

Parametric polymorphism provides a tool to write reusable and modular programs. Assignment on locations provides an effective and efficient way to program. The combination of parametric polymorphism and assignment promises a great tool for programming. A large amount of work, most noticeably, the ML family of programming languages, covers the combination.

Unfortunately, the combination is not very straightforward. Based on a purely functional setting, polymorphism allows one term to be used with different types. In an imperative setting, the uses of one identifier (term variable) usually refer to one fixed memory location. As one fixed memory location commonly cannot accommodate different types, the two ideas of polymorphism and inplace update cannot freely coexist without compromising type safety. The following lines of code (in an OCAML syntax [51]) help to illustrate the point further [1]:

$$\textbf{let}\ x = \mathsf{ref}\ [\,]\ \textbf{in}$$
$$x := \mathsf{true} :: !x;$$
$$(\mathsf{List.hd}\ !x) + 9;;$$

The **let** expression binds x to a reference cell initialized to an empty list of polymorphic type list of X. If the polymorphism is unconstrained, then X can be instantiated to different types. The assignment in the body instantiates X with boolean type while the addition in the body (in the third line of the code) instantiates X with integer type. Evaluating the combination will result in a run time type exception when the boolean true is added to the integer 9.

The problem is due to having some updatable data being polymorphic. However, there is no pressing need for any updatable data to be polymorphic. In a functional language based on λ-calculus [15], every term has the nature of functions so they are all potentially polymorphic. Since the expressive power of polymorphism in a functional language is all in polymorphic functions; the existence of polymorphic data structures, such as the empty

[1]In OCAML, ref := and ! are three operators for creating, updating and getting the value of reference cells.
The semi-colon punctuation (;) is the infix operator for sequential composition.
The double-colon (::) is the infix operator for creating a list from a head and a tail, List.hd is a function return the head of non-empty list.

list, is incidental. From an imperative viewpoint, functions are different from data and inplace assignment acts only on data. Combining the two ideas, we can set up a natural boundary between (potentially) polymorphic functions and updatable data structures. Hence, to solve the typing problem of combining parametric polymorphism and updatable data it suffices to separate functions and data in the type system.

3.2 Value Polymorphism

In this section, we introduce a modification of the type rules to naturally support parametric polymorphism and imperative features. The rules supporting polymorphism are those for typing **let**-terms and fixpoints.

The **let** typing rule of the original pattern calculus (in Figure 2.2):

$$\frac{\Delta,\Delta_1;\Gamma \vdash s : S \quad \Delta;\Gamma, x : \forall\Delta_1.S \vdash t : T}{\Delta;\Gamma \vdash \textbf{let}\ x = s\ \textbf{in}\ t : T}$$

gives the variable x a polymorphic type by using the type scheme $\forall\Delta_1.S$.

We are going to restrict the use of this rule; for example when s is an extension (known to be function). For the default case, we introduce a new monomorphic **let** rule that simply gives x the monomorphic type (scheme) S (without quantification).

$$\frac{\Delta;\Gamma \vdash s : S \quad \Delta;\Gamma, x : S \vdash t : T}{\Delta;\Gamma \vdash \textbf{let}\ x = s\ \textbf{in}\ t : T}$$

Term contexts (Γ)

$$\frac{\Delta \vdash}{\Delta;\, \vdash} \qquad \frac{\Delta;\Gamma \vdash \quad \Delta \vdash_s \tau}{\Delta;\Gamma, x:\tau \vdash}\; x \notin \Gamma$$

Patterns (p)

$$\frac{}{X; x:X \vdash_o x:X} \qquad \frac{}{\Delta;\, \vdash_o c:T}\; c:\forall\Delta.T$$

$$\frac{\Delta;\Gamma \vdash_o p:T \quad \Delta_1;\Gamma_1 \vdash_o p_1:T_1 \quad \Delta,\Delta_1,X;\Gamma,\Gamma_1 \vdash}{\Delta,\Delta_1; v\Gamma, v\Gamma_1 \vdash_o p\; p_1 : vX}\; v=\mathcal{U}(T_1 \rightarrow X, T)$$

Terms (s,t)

$$\frac{\Delta;\Gamma \vdash}{\Delta;\Gamma \vdash x:\sigma T}\; \begin{array}{l}\Gamma(x)=\forall\Delta_1.T\\ \sigma:\Delta_1 \rightarrow \Delta\end{array}$$

$$\frac{\Delta;\Gamma \vdash}{\Delta;\Gamma \vdash c:\sigma T}\; \begin{array}{l}c:\forall\Delta_1.T\\ \sigma:\Delta_1 \rightarrow \Delta\end{array}$$

$$\frac{\Delta;\Gamma \vdash s:T \rightarrow S \quad \Delta;\Gamma \vdash t:T}{\Delta;\Gamma \vdash s\; t:S}$$

$$\frac{\Delta;\Gamma \vdash t:T \rightarrow S \;\; \Delta_1;\Gamma_1 \vdash_o p:T_1 \;\; \Delta,\Delta_1; v\Gamma, v\Gamma_1 \vdash s:vS}{\Delta;\Gamma \vdash \mathbf{at}\; p\; \mathbf{use}\; s\; \mathbf{else}\; t : T \rightarrow S}\; v=\mathcal{U}(T_1,T)$$

$$\frac{\Delta;\Gamma \vdash t:T \rightarrow S \quad \Delta_1;\Gamma_1 \vdash_o p:T_1 \quad \Delta,\Delta_1;\Gamma,\Gamma_1 \vdash}{\Delta;\Gamma \vdash \mathbf{at}\; p\; \mathbf{use}\; s\; \mathbf{else}\; t : T \rightarrow S}\; \mathcal{U}(T_1,T)\uparrow$$

$$\frac{\Delta,\Delta_1;\Gamma \vdash s:S \quad \Delta;\Gamma, x:\forall\Delta_1.S \vdash t:T}{\Delta;\Gamma \vdash \mathbf{let}\; x=s\; \mathbf{in}\; t:T}\; s \text{ is non-expansive}$$

$$\frac{\Delta;\Gamma \vdash s:S \quad \Delta;\Gamma, x:S \vdash t:T}{\Delta;\Gamma \vdash \mathbf{let}\; x=s\; \mathbf{in}\; t:T}$$

$$\frac{\Delta,\Delta_1;\Gamma, x:\forall\Delta_1.T \vdash t:T}{\Delta,\Delta_1;\Gamma \vdash \mathbf{fix}\; (x,t):T}\; t \text{ is non-expansive}$$

$$\frac{\Delta;\Gamma, x:T \vdash t:T}{\Delta;\Gamma \vdash \mathbf{fix}\; (x,t):T}$$

Figure 3.1: The Pattern Calculus Restricted to Value Polymorphism

This idea is the same as the "value polymorphism" approach [69] (reviewed in Chapter 1) that limits polymorphism for only non-expansive terms. We will use the same term here. A term is called *non-expansive* if it is an extension or a variable. As discussed above, we could define non-expansive to be extensions only but the addition of variables is proved to be convenient without compromising safety. In fact, one could try to add more terms to be non-expansive but we did not find any convincing need for more.

In the expression **let** $x = s$ **in** t, if s is non-expansive, then x is treated polymorphically in t, in the default case, x is treated monomorphically in t.

$$\frac{\Delta, \Delta_1; \Gamma \vdash s : S \quad \Delta; \Gamma, x : \forall \Delta_1 . S \vdash t : T}{\Delta; \Gamma \vdash \mathbf{let}\ x = s\ \mathbf{in}\ t : T}\ s \text{ is non-expansive}$$

A similar modification is for the rule for fix-points. One polymorphic rule for **fix** from the original pattern calculus now becomes two rules. The polymorphic one is used only for non-expansive terms:

$$\frac{\Delta, \Delta_1; \Gamma, x : \forall \Delta_1 . T \vdash t : T}{\Delta, \Delta_1; \Gamma \vdash \mathbf{fix}\ (x, t) : T}\ t \text{ is non-expansive}$$

The default is used the monomorphic rule:

$$\frac{\Delta; \Gamma, x : T \vdash t : T}{\Delta; \Gamma \vdash \mathbf{fix}\ (x, t) : T}$$

In most, as functions can be used polymorphically, so can data structure. Variables bound to data are free to be used polymorphically. For example, OCAML is able to type the following lines of code

$$\textbf{let } x = [\,] \textbf{ in } (0 :: x, \mathsf{true} :: x)$$

$$\textbf{let } x = [\,] \textbf{ in } x :: x$$

in which the empty list is used polymorphically by defining it to be one of the values. We could do the same here in our system by declaring the empty list (or a class of term containing the empty list) to be non-expansive. However, we choose not to do so, and this is no loss. Data (non-function terms) with the potential of having more than one type usually contains little real information, such as the nil list or vector of length zero. In our viewpoint, limiting the polymorphism on data does not reduce the expressive power of the language in any significant way. By giving up polymorphism only on non-function terms, we can add the imperative features into the system in a simple and fully type safe way.

3.3 Conclusion

Parametric polymorphism can be safely integrated into an imperative system in a simple way. The key point is to separate extensions (functions) which are potentially polymorphic from updatable terms. We use the polymorphic let rule for non-expansive terms only, and use the monomorphic let rule for other terms. The two rules for let expressions (and the two for fix-points) are the way to combine polymorphic functions and monomorphic updatable data into one small simple system.

In this chapter, we apply value polymorphism to the pattern calculus.

In fact, we have not yet introduced any complete system with both of these features, i.e. we have not introduced any formal definition of location nor update. The reason is that we do not yet want to commit the idea into a specific system. As a result, it can be used in different systems in flexible ways. In the next two chapters, one can see how the idea can be used for systems where locations are built in a functional way as well as in an imperative way.

In the research, I spent some times exploring alternative approaches to value polymorphism. One interesting observation is that the typing might be used to separate functions which are potentially polymorphic from updatable terms. The combinatory type provides a sound foundation for the separation. I would have worked on this approach for longer if time permitted. With some small modification, the new idea can be used in other combinations of parametric polymorphism and updatable locations. In particular, one might modify ML to use it so that the imperative features can be contained in a clearer and more elegant type system.

Chapter 4

Location Types

4.1 Introduction

One of the great strengths of functional programming languages is that it relieves the programmer of the need to manage memory, which helps make programs shorter and easier to reason about. The price to be paid is that their compilers must take a conservative approach to memory allocation, often allocating new heap-space and garbage collecting the old values when inplace update would have been perfectly safe.

A significant effort has been made to improve the efficiency of this process. For example, types in compilation [6, 8, 10, 20, 68] uses type information to ensure the safety of some inplace updating. This works well for integers, floats and tuples built of such simple types, but does not handle recursive types such as lists where the type does not determine the shape of its values. Monitoring list lengths etc. may be attempted using *sized types* [25, 53] or other dependently-typed systems, e.g. [72] with the aim of extracting compiletime information.

This chapter adds imperative features to the pattern calculus. All the theory in this chapter is backed up by an implementation which is a version of FISh2 with imperative features. There are some choices in designing an imperative system. The most important choice for the system in this chapter is that there will be a separate class of location types which are distinct from the existing functional types as in ML. This choice gives the name of this chapter; the next chapter will explore an alternative in which locations share the same type with their values.

The system is built in two steps. First, we add to the pattern calculus the notion of atomic locations and operations on them. Second, we introduce *location constructors*. Using location constructors, *constructed locations* are built in the same way as constructed data are built in the pattern calculus. Generic operations on locations using path polymorphism can be defined by pattern-matching on these location constructors. Other imperative features: while-loops and for-loops; vector types; generic output are also added to the system to improve usability.

Using path polymorphism from the pattern calculus, we extend primitive assignment on locations to generic assignment on constructed locations. Defining generic assignment in this way, inplace update is used whenever it is reasonable to do so, based on matching the structure of the location with that of its new value. Of course, the inplace update cannot be applied when the structures (shapes) are different. We suggest the use of locations of locations - which are called references in this chapter, to be assured of safety when the inplace update is not appropriate. One of the main aims of this chapter is to allow assignments to determine at run-time whether to assign inplace or not.

Two sets of examples are provided. Those in Section 4.5 show that the system has all the expressive power of the imperative features in systems such that ML. Those in Section 4.9 illustrate some of the novel expressive power of the system. Bubble-sort shows how a space efficient sorting program can be written using higher-order functions and pattern-matching. The program converge shows how to iterate a function on a data structure while using space efficiently.

4.2 Locations

This section adds to the pattern calculus some primitive imperative features similar to the style of ML [63]. The features include: a command type; location types; the skip constant; and primitive operations to create new locations, to get the values of locations and to assign to locations. A key point is that the primitive assignment is treated as an atomic built-in operation, simple to describe but extravagant with space in some cases.

The work in this section is based on the pattern calculus with value polymorphism as described in the previous chapter. The definition of types, terms and type derivation rules are unchanged.

First, the new type constant

$$\mathsf{comm}$$

is equipped with a term constant

$$\mathsf{skip} : \mathsf{comm}.$$

The execution of the command skip has no (side) effect. We decide that skip is a constructor so that it can be pattern-matched. One might choose to identify the type unit with the command type (and identify the constructor un with skip). However, our choice here is to make them distinct so that the addition of imperative features is separated from the representation of data.

One might choose to have sequential composition as a new term constant of the system. However, our choice is to define sequential composition as an extension to reduce the number of new constants in the system:

$$(\mathsf{seq} : \mathsf{comm} \to X \to X) = \;\mid \mathsf{skip} \to (\;\mid y \to y)\;.$$

We may use the usual syntactic sugar $x; y$ for seq x y.

Next, considering assignable locations, each type T has the associated *location type*

$$\mathsf{loc}\; T$$

of *locations* that store (assignable) values of type T. There is no restriction on the type T so we can have locations that hold any terms, including other locations and functions e.g. loc int, loc (list int), loc (loc int) and loc (int $\to$ int) are all types.

We also introduce three new constants for creating a new location with a given value; getting the value from a location; and updating the value of a location:

$$\begin{aligned}\mathsf{primloc} &: \forall X.X \to \mathsf{loc}\ X \\ \mathsf{primval} &: \forall X.\mathsf{loc}\ X \to X \\ \mathsf{primassign} &: \forall X.\mathsf{loc}\ X \to X \to \mathsf{comm}.\end{aligned}$$

A term of the form primloc t creates a new location whose initial value is that of t. A term of the form primval t represents the value stored at the location t. A term of the form primassign t t' updates the location t with the value of t'. The calculus does not specify how locations are to be deleted; this must be handled elsewhere e.g. by some form of garbage collection [11].

We also introduce a new constant for printing constructors (especially primitive datum constructors)

$$\mathsf{output\_basic} : \forall X.X \to \mathsf{comm}.$$

Detail description about the function is given later in this chapter when discussing about generic output.

4.3 Evaluation

This section describes the evaluation rules of the system.

The *values* (meta-variable v) are given by:

$$v ::= x \mid \textbf{at}\ p\ \textbf{use}\ s\ \textbf{else}\ t \mid b\ v_1\ \ldots\ v_k\ \{k < \text{arity}(b)\ \textbf{if}\ b \text{ is an operator}\}.$$

That is, a value is either: a term variable; an extension; a constructed term whose arguments are all values; or a partially-applied operator whose arguments are all values. For example, primassign x and primplusint $<9>$ are values but primassign x 9 and primplusint $<9>$ $<5>$ are not.

A *store* (meta-variable Σ) is a partial function from term variables to values. We use the syntax $\Sigma, x \mapsto v$ to update the store Σ with a new mapping $x \mapsto v$ (if x is already in the domain of Σ then the new value replaces the old one). We use dom Σ for the domain of a store Σ.

An *evaluation context* (Σ, t) is a pair of a store Σ and a term t. Evaluation employs a big-step operational semantics [65], where evaluating to a value is defined directly from the *evaluation rules.* Evaluation rules are expressed using judgements of the form

$$(\Sigma, t) \Rightarrow (\Sigma', v)$$

where (Σ, t) and (Σ', v) are evaluation contexts and v is a value.

The evaluation rules are given in Figure 4.1. Note that the order of the rules is significant and these rules need to be applied in order. The significance of the order in evaluation rules makes two reduction rules for pattern-match failure to be handled by one evaluation rule. Note that the store gets a passive role in the first ten rules. Most of these rules are derived from the reduction rules of the Pattern Calculus (Figure 2.3). If $t > t'$ is such a rule, then as a first approximation we have the evaluation rule

$$\frac{(\Sigma, t') \Rightarrow (\Sigma', v)}{(\Sigma, t) \Rightarrow (\Sigma', v)}$$

Rule 1 $$\frac{(\Sigma, t[\mathbf{fix}\ (x,t)/x]) \Rightarrow (\Sigma', v)}{(\Sigma, \mathbf{fix}\ (x,t)) \Rightarrow (\Sigma', v)}$$

Rule 2 $$\frac{(\Sigma, s) \Rightarrow (\Sigma', v_0) \quad (\Sigma', t[v_0/x]) \Rightarrow (\Sigma'', v_1)}{(\Sigma, \mathbf{let}\ x = s\ \mathbf{in}\ t) \Rightarrow (\Sigma'', v_1)}$$

Rule 3 $$\frac{(\Sigma, s[v/x]) \Rightarrow (\Sigma', v_1)}{(\Sigma, (\mathbf{at}\ x\ \mathbf{use}\ s\ \mathbf{else}\ t)\ v) \Rightarrow (\Sigma', v_1)}$$

Rules 4,5 $$\frac{(\Sigma, s) \Rightarrow (\Sigma', v)}{(\Sigma, (\mathbf{at}\ c\ \mathbf{use}\ s\ \mathbf{else}\ t)\ c) \Rightarrow (\Sigma', v)}$$

$$\frac{(\Sigma, (\mathbf{at}\ p_1\ \mathbf{and}\ p_2\ \mathbf{use}\ s\ \mathbf{else}\ t)\ t_1\ t_2) \Rightarrow (\Sigma', v)}{(\Sigma, (\mathbf{at}\ p_1\ p_2\ \mathbf{use}\ s\ \mathbf{else}\ t)\ (t_1\ t_2)) \Rightarrow (\Sigma', v)} \quad \begin{array}{l} t_1\ t_2 \text{ is a value} \\ t_1 \text{ is a constructed term} \end{array}$$

Rule 6 $$\frac{(\Sigma, t\ v) \Rightarrow (\Sigma', v_1)}{(\Sigma, (\mathbf{at}\ p\ \mathbf{use}\ s\ \mathbf{else}\ t)\ v) \Rightarrow (\Sigma', v_1)}$$

Rule 7 $$\frac{}{(\Sigma, v\ v_1) \Rightarrow (\Sigma, v_1)} \quad \begin{array}{l} v \text{ headed by an operator} \\ v_1 \text{ headed by } \mathsf{exn} \end{array}$$

Rule 8 $$\frac{}{(\Sigma, d\ d_0 \cdots d_{n-1}) \Rightarrow (\Sigma, d_n)}\ d\ d_0\ \cdots\ d_{n-1} = d_n$$

Rule 9 $$\frac{}{(\Sigma, \mathsf{primequal}\ c\ c) \Rightarrow (\Sigma, \mathsf{true})}$$

Rule 10 $$\frac{}{(\Sigma, \mathsf{primequal}\ v_1\ v_2) \Rightarrow (\Sigma, \mathsf{false})}$$

Rules 11,12,13 $$\frac{}{(\Sigma, \mathsf{primloc}\ v) \Rightarrow (\Sigma, u \mapsto v, u)}\ u \text{ fresh}$$

$$\frac{}{(\Sigma, \mathsf{primval}\ u) \Rightarrow (\Sigma, \Sigma(u))}\ u \in \mathsf{dom}\Sigma \qquad \frac{}{(\Sigma, \mathsf{primval}\ v) \Rightarrow (\Sigma, \mathsf{exn}\ v)}$$

Rule 14 $$\frac{}{(\Sigma, \mathsf{primassign}\ u\ v) \Rightarrow (\Sigma, u \mapsto v, \mathsf{skip})}\ u \in \mathsf{dom}\Sigma$$

Rule 15 $$\frac{}{(\Sigma, \mathsf{primassign}\ v_0\ v_1) \Rightarrow (\Sigma, \mathsf{exn}\ v_0)}$$

Rule 16 $$\frac{}{(\Sigma, \mathsf{output\_basic}\ v) \Rightarrow (\Sigma, \mathsf{skip})}\ \text{output } v \text{ if } v \text{ is a constructor}$$

Rules 17,18 $$\frac{}{(\Sigma, v) \Rightarrow (\Sigma, v)}$$

$$\frac{(\Sigma, t_0) \Rightarrow (\Sigma', v_0) \quad (\Sigma', t_1) \Rightarrow (\Sigma'', v_1) \quad (\Sigma'', v_0\ v_1) \Rightarrow (\Sigma''', v_2)}{(\Sigma, t_0\ t_1) \Rightarrow (\Sigma''', v_2)} \quad \begin{array}{l} t_0, t_1 \text{ are not} \\ \text{both values} \end{array}$$

Figure 4.1: Evaluation Rules with Location Types

The correct interpretation typically involves evaluation of sub-terms too.

The first six rules involve evaluation of let-terms, fix-terms and applications of extensions to terms. The seventh rule is the standard approach to evaluate an application to an exception. The eighth rule summarizes the evaluation of fully-applied datum operators. The next two rules are to evaluate the applications of primequal to two terms.

The next five rules are to evaluate terms involving the imperative constants primloc, primval and primassign which may involve modifying and looking up the store.

The last two rules are standard rules about the evaluation of value and application of terms. A value is evaluated to itself. An application $t_0\ t_1$ is evaluated by first evaluating t_0 and then evaluating t_1 follow by evaluating the application of the two results.

The evaluation rules do not constrain the implementation of storage operations. One expects that assignment of datum values such as integers or floats will be performed inplace and that assignment of functions will be by allocating fresh memory. The delicate case is an assignment of structured data, such as a list. This will be addressed in sub-section 4.5.1.

4.4 Properties of Evaluation

We will show that evaluation preserves typing. To do so, it requires agreement between the types of variables as understood in the term context and the types of their corresponding value in the store.

When referring to some particular known store Σ, we call a term variable a *store variable* (meta-variable u) if it is in the domain of Σ. We decide that all store variables have monomorphic (trivial) type schemes.

We define a new property about the relation between the typing of the store variables and of their values. The judgement $\Delta; \Gamma \vdash_\Sigma \Sigma$ asserts that the store Σ is *well-typed* in context $\Delta; \Gamma$. For a store Σ with n variables $\{u_1 \mapsto v_1, \ldots, u_n \mapsto v_n\}$, $\Delta; \Gamma \vdash_\Sigma \Sigma$ means that $\Delta; \Gamma \vdash$ and $\Delta; \Gamma \vdash v_i : T_i$. In other words, in the type context $\Delta; \Gamma$, a store Σ is called well-typed if the domain of Σ is a subset of the domain of Γ and each store variable in Γ must be of the location type corresponding to the type of its value in Σ. Note that there is no restriction on using a store variable in its value or in value of another store variable. That allows the ability to define recursive and mutually recursive locations in the system. An example of recursive locations is in Sub-Section 4.5.2.

Note that the type context has to expand during the evaluation to cope with the new store variables. In particular, the evaluation of $\mathsf{primloc}\ v$ results in a new store variable u with the associate initial value v. To type the result of the evaluation, we need to expand the type context with the new term variable u. The type (scheme) of a store variable u is initialized to be the type derived from the type of its initial value v. With the evaluation rule

$$\frac{}{(\Sigma, \mathsf{primloc}\ v) \Rightarrow (\Sigma, u \mapsto v, u)}\ u \text{ fresh}$$

the type scheme of u is given by the location type of the type of v.

$$\frac{\Delta;\Gamma \vdash_{\Sigma} \Sigma \quad \Delta;\Gamma \vdash v : T}{\Delta;\Gamma, u : \mathsf{loc}\ T \vdash_{\Sigma} \Sigma, u \mapsto v}\ u \text{ fresh}$$

By that choice of extending the type context, if the store is well-typed before, then clearly it remains well-typed after the evaluation of $\mathsf{primloc}\ v$.

We introduce a new form of typing rule for an evaluation context. The judgement $\Delta;\Gamma \vdash_i (\Sigma, t) : T$ asserts that the store is well-typed ($\Delta;\Gamma \vdash_{\Sigma} \Sigma$) and t has type T in the context of $\Delta;\Gamma$ ($\Delta;\Gamma \vdash t : T$).

Theorem 2 *Evaluation preserves typing. That is, if* $\Delta;\Gamma \vdash_i (\Sigma, t) : T$ *and* $(\Sigma, t) \Rightarrow (\Sigma', v)$ *then there is a context* Γ_1 *such that* $\Delta;\Gamma,\Gamma_1 \vdash_i (\Sigma', v) : T$.

Proof: The proof is by induction on the structure of the evaluation of (Σ, t). In all the cases from Figure 4.1, apart from those involving primval, primloc and primassign, the store has a passive role. So, the proof corresponds to the cases for the proof of subject reduction in the pattern calculus [35]. Note that the proof uses the substitution lemma, (Chapter 2, Lemma 1). We include here the proofs for Rules 1 and 2, as the typing rules of these cases have been changed by "value polymorphism", Rules 11,12 and 14, as these involve the store.

Rule 1

$$\frac{(\Sigma, t[\mathbf{fix}\ (x,t)/x]) \Rightarrow (\Sigma', v)}{(\Sigma, \mathbf{fix}\ (x,t)) \Rightarrow (\Sigma', v)}$$

From the assumption $\Delta;\Gamma \vdash_i (\Sigma, \mathbf{fix}\ (x,t)) : T$ it follows that $\Delta;\Gamma \vdash_{\Sigma} \Sigma$ and $\Delta;\Gamma \vdash \mathbf{fix}\ (x,t) : T$. There are two possible cases for typing of fix-points.

If the polymorphic **fix** rule is used, there must be a derivation $\Delta_2, \Delta_1; \Gamma, x : \forall\Delta_1.T \vdash t : T$ where $\Delta_2, \Delta_1 = \Delta$. Let σ be a renaming substitution such that $\sigma : \Delta_1 \rightarrow \Delta_3$ and $\Delta, \Delta_3; \Gamma$ is well defined. We can deduce that $\Delta; \Gamma, x : \forall\Delta_3.T \vdash t : \sigma T$ and $\Delta, \Delta_3; \Gamma \vdash \mathbf{fix}\ (x, t) : T$. Applying the substitution lemma we have $\Delta, \Delta_3; \Gamma \vdash t[\mathbf{fix}\ (x,t)/x] : \sigma T$. Renaming the variable in Δ_3 back to Δ_1 by σ^{-1} we have $\Delta_2, \Delta_1; \Gamma \vdash t[\mathbf{fix}\ (x,t)/x] : \sigma^{-1}\sigma T$. So now we know that $\Delta; \Gamma \vdash_i (\Sigma, t[\mathbf{fix}\ (x,t)/x]) : T$. Combining it with the induction hypothesis on the evaluation of $t[\mathbf{fix}\ (x,t)/x]$, we know that there is Γ_1 such that $\Delta; \Gamma, \Gamma_1 \vdash_i (\Sigma', v) : T$.

If the monomorphic **fix** rule is used, there must be a derivation $\Delta; \Gamma, x : T \vdash t : T$. Applying the substitution lemma we have $\Delta, \Gamma \vdash t[\mathbf{fix}\ (x,t)/x] : T$. Combining it with the induction hypothesis on the evaluation of $t[\mathbf{fix}\ (x,t)/x]$, we know that there is Γ_1 such that $\Delta; \Gamma, \Gamma_1 \vdash_i (\Sigma', v) : T$.

Rule 2

$$\frac{(\Sigma, s) \Rightarrow (\Sigma', v_0) \quad (\Sigma', t[v_0/x]) \Rightarrow (\Sigma'', v_1)}{(\Sigma, \mathbf{let}\ x = s\ \mathbf{in}\ t) \Rightarrow (\Sigma'', v_1)}$$

From the assumption $\Delta; \Gamma \vdash_i (\Sigma, \mathbf{let}\ x = s\ \mathbf{in}\ t) : T$ it follows that $\Delta; \Gamma \vdash_\Sigma \Sigma$ and $\Delta; \Gamma \vdash \mathbf{let}\ x = s\ \mathbf{in}\ t : T$. There are two possible cases for typing of let-expressions.

If the polymorphic **let** rule is used, then s is non-expansive. From the definition of non-expansive we know $(\Sigma, s) \Rightarrow (\Sigma, s)$ so Σ' is

Σ and v_0 is s. From the polymorphic **let** rule, there are some Δ_1 such that $\Delta, \Delta_1; \Gamma \vdash s : S$ and $\Delta; \Gamma, x : \forall\Delta_1.S \vdash t : T$. Using the substitution lemma for t and s we know that $\Delta, \Delta_1; \Gamma \vdash t[s/x] : T$. Since Δ_1 is not needed for typing T (in the type of t) we have $\Delta, \Gamma \vdash t[s/x] : T$. That implies $\Delta; \Gamma \vdash_i (\Sigma, t[s/x]) : T$. Applying the induction hypothesis to the evaluation of $(\Sigma', t[v_0/x])$ we can conclude that there is Γ_1 such that $\Delta; \Gamma, \Gamma_1 \vdash_i (\Sigma'', v_1) : T$.

If the monomorphic **let** rule is used, we have $\Delta; \Gamma \vdash s : S$ and $\Delta; \Gamma, x : S \vdash t : T$ and so $\Delta; \Gamma \vdash_i (\Sigma, s) : S$. Applying the induction hypothesis to the evaluation of s, we can conclude that there is Γ_1 such that $\Delta; \Gamma, \Gamma_1 \vdash_i (\Sigma', v_0) : S$. So we have $\Delta; \Gamma, \Gamma_1 \vdash_\Sigma \Sigma'$. Moreover, using the substitution lemma for t and v_0 we know that $\Delta; \Gamma, \Gamma_1 \vdash t[v_0/x] : T$. That implies $\Delta; \Gamma, \Gamma_1 \vdash_i (\Sigma', t[v_0/x]) : T$. Applying the induction hypothesis to the evaluation of $(\Sigma', t[v_0/x])$ we can conclude that there is Γ_2 such that $\Delta; \Gamma, \Gamma_1, \Gamma_2 \vdash_i (\Sigma'', v_1) : T$.

Rule 11

$$\frac{}{(\Sigma, \mathsf{primloc}\ v) \Rightarrow (\Sigma, u \mapsto v, u)}\ u \text{ fresh}$$

From the assumption $\Delta; \Gamma \vdash_i (\Sigma, \mathsf{primloc}\ v)) : T$ we know that Σ is a well-typed store, and that T is of the form $\mathsf{loc}\ T'$ for some T' which is a type for v. With the typing rule for new store variable (u), we have $\Sigma, u \mapsto v$ is also well-typed store in the context $\Delta; \Gamma, u : \mathsf{loc}\ T'$ and u can take the type $\mathsf{loc}\ T'$ and so $\Delta; \Gamma, u : T \vdash_i (\Sigma, u \mapsto v, u) : T$

Rule 12

$$\frac{}{(\Sigma, \mathsf{primval}\ u) \Rightarrow (\Sigma, \Sigma(u))}\ u \in \mathsf{dom}\ \Sigma$$

From the assumption that Σ is a well-typed store and the typing of primval the claim holds.

Rule 14

$$\frac{}{(\Sigma, \mathsf{primassign}\ u\ v) \Rightarrow (\Sigma, u \mapsto v, \mathsf{skip})}\ u \in \mathsf{dom}\ \Sigma$$

From the assumption that Σ is a well-typed store it follows that $\Delta; \Gamma \vdash u : \mathsf{loc}\ T$ and u is in the domain of Σ since Σ is well-typed. From the well-typedness of primassign $u\ v$ we can deduce that $\Delta; \Gamma \vdash v : T$. So the updated store $\Sigma, u \mapsto v$ remains well-typed. Moreover, the types of primassign $u\ v$ and skip are both comm, so the claim holds.

□

A store Σ is called *closed* if for all variables u in the domain of Σ, all free variables of $\Sigma(u)$ are in the domain of the store Σ. The evaluation context (Σ, t) is called *closed* if Σ is closed and all free variables of the term t are in the domain of Σ.

Lemma 3 *Evaluation preserves closedness. That is, suppose there is an evaluation $(\Sigma, t) \Rightarrow (\Sigma', v)$ and (Σ, t) is closed then (Σ', v) is also closed.*

Proof: The proof is by induction on the structure of evaluation.

We do a case analysis on the evaluation rules in Figure 4.1. Most of the cases are straight forward using the induction hypothesis. The more interesting cases are for Rules 1,2,11 and 12. Let D be the set of all term variables in the domain of Σ.

Rule 1

$$\frac{(\Sigma, t[\mathbf{fix}\ (x,t)/x]) \Rightarrow (\Sigma', v)}{(\Sigma, \mathbf{fix}\ (x,t)) \Rightarrow (\Sigma', v)}$$

From the assumption about the closedness of $(\Sigma, \mathbf{fix}\ (x,t))$ it follows that $fv(t) - \{x\} \subset D$. We can deduce that $(\Sigma, t[\mathbf{fix}\ (x,t)/x])$ is also closed. Using that with the induction hypothesis on the evaluation of $(\Sigma, t[\mathbf{fix}\ (x,t)/x])$ we can conclude that (Σ', v) is closed.

Rule 2

$$\frac{(\Sigma, s) \Rightarrow (\Sigma', v_0) \quad (\Sigma', t[v_0/x]) \Rightarrow (\Sigma'', v_1)}{(\Sigma, \mathbf{let}\ x = s\ \mathbf{in}\ t) \Rightarrow (\Sigma'', v_1)}$$

From the assumption about the closedness of $(\Sigma, \mathbf{let}\ x = s\ \mathbf{in}\ t)$ it follows that (Σ, s) is closed. Applying the induction hypothesis on the evaluation of (Σ, s) we have (Σ', v_0) is closed. So $(\Sigma', t[v_0/x])$ is closed and applying the induction hypothesis on the evaluation of that term we have (Σ'', v_1) is closed.

Rule 11

$$\frac{}{(\Sigma, \mathsf{primloc}\ v) \Rightarrow (\Sigma, u \mapsto v, u)}\ u \text{ fresh}$$

From the assumption about the closedness of $(\Sigma, \mathsf{primloc}\ v)$, all free variables of v are in the domain of Σ and since u is a term variable in the domain of the store $\Sigma, u \mapsto v$ so the evaluation context $(\Sigma, u \mapsto v, u)$ remains closed.

Rule 12

$$\frac{}{(\Sigma, \mathsf{primval}\ u) \Rightarrow (\Sigma, \Sigma(u))}\ u \in \mathsf{dom}\Sigma$$

From the assumption that $(\Sigma, \mathsf{primval}\ u)$ is closed, and u is a term variable in the domain of Σ, we know that all free variables in $\Sigma(u)$ are in the domain of Σ so $(\Sigma, \Sigma(u))$ is closed.

□

Combining this above lemma with the previous theorem we can claim that evaluation preserves well-typedness and closedness. A closed and well-typed term is called a *program*. The following theorem discusses the property of evaluation on programs.

Theorem 4 *Evaluation of programs never gets stuck. That is, if we have a derivation* $\Delta; \Gamma \vdash_i (\Sigma, t) : T$ *and* (Σ, t) *is closed then there is an evaluation rule where the left hand side of the conclusion matches with* (Σ, t).

Proof: Note that although the order of evaluation rules is significant, we need only find a single rule that can be applied.

The proof is by case analysis on the structure of t, and is mostly based on the definition of values and the evaluation rules in Figure 4.1.

When t is a variable, constant, or extension then t is a value. We could always apply the second last rule which states that any value evaluates to itself.

When t is a fix point construction, then the first rule applies.

When t is a let-expression, then the second rule applies.

The only case left is when t is an application. As one could try and apply the last rule when at least one of the parts of the application is not a value, without loss of generality, now we can assume that t has the form v_0 v_1 where v_0 and v_1 are values. Now consider the cases for the choices of v_0.

If v_0 is a variable, then the closed condition of the evaluation context forces v_0 to be a variable in the domain of the store and the well-typedness of the store forces it to be of location type. However, the function part of an application (always of function type) can never be of location type, therefore this case never happens.

If v_0 is an extension, then Rule 6 applies.

If v_0 is headed by a constant b then t is also headed by b. By the well-typedness of t and the definition of values (as v_0,v_1 are values), there are two cases of such a term t. If t is a value, we can apply the rule which states that any value evaluates to itself. Alternatively, if t is a fully-applied operator whose arguments are all values, we can apply one of the Rules 7,8,10,11,13,15 and 16.

□

4.5 Examples for Primitive Imperative Features

This chapter as presented so far is an expansion of the pattern calculus with primitive imperative features. This section contains some small examples for that expansion. The main aim of the examples is to illustrate that the imperative features behave in the usual ways such that those of ML.

4.5.1 Updating Lists

Previously, we have defined a type of list with syntactic sugar $[1, 2, 3]$ for the list of three number 1, 2 and 3. Now consider the example of creating and updating lists

$$\begin{aligned}&\textbf{let } x = \mathsf{primloc}\ [1]\ \textbf{in}\\ &\mathsf{primassign}\ x\ [2];\\ &\mathsf{primassign}\ x\ [8, 9].\end{aligned}$$

Clearly, the first assignment could be inplace, but any simple implementation of primassign will miss this opportunity since it will not be able to distinguish this case from the second, shape-changing assignment.

4.5.2 Linked-Lists

This subsection defines a typical imperative structure, linked-list:

$$\mathsf{type\ link}\ X = \mathsf{link}\ \textbf{of}\ X\ \textbf{and}\ \mathsf{loc}\ (\mathsf{link}\ X).$$

A linked-list of type X contains a term of type X and a location linked to another linked-list of X. Some examples of linked-lists are:

$$\begin{aligned}l_1 &= \mathsf{primloc}\ (\mathsf{link}\ 1\ \mathsf{exn})\\ l_2 &= \mathsf{primloc}\ (\mathsf{link}\ 2\ l_1)\end{aligned}$$

One can change l_1 into a recursive (self linked) list by:

$$\mathsf{primassign}\ l_1\ (\mathsf{link}\ 3\ l_1).$$

In functional terms, l_1 is now a list holding an infinite number of integers 3.

4.6 Constructed Locations

This section introduces *location constructors* into the system. With location constructors, structured locations are built out of smaller locations in the same way constructed terms are built. Using pattern-matching on location constructors, generic functions for locating, valuing and assigning are defined in the source code based on their primitive versions, just as the generic function plus is based on datum addition. Inplace update is possible when the structure of the location is matched by that of its new value. This matching is based on comparing constructors.

For pattern-matching against locations, we introduce a new constructor for creating location type:

$$\mathsf{conloc}\ : \forall X.X \rightarrow \mathsf{loc}\ X.$$

Note that even though the two constants primloc and conloc have the same type, their roles are quite separate. primloc is not a constructor but a built-in term for creating directly updatable locations (associated with some variable of the store). conloc is a constructor to be applied to other constructors.

A term built by conloc is a constructed term that can be used in pattern-matching and it does not directly associate with any variable of the store. Hence, the use of primval or primassign on a term constructed by conloc results in an exception.

As larger terms are built by application of smaller terms, for constructing larger locations, we introduce a new built-in constructor for converting a location of an application to the application of locations:

$$\mathsf{locap}\ : \forall X, Y.\mathsf{loc}\ (X \to Y) \to \mathsf{loc}\ X \to \mathsf{loc}\ Y.$$

For each constructor c of type $X \to Y$, we will create the corresponding function of type loc $X \to$ loc Y in two steps: the first step is to create the location constructor conloc c of type loc $(X \to Y)$; the second step is to use locap to convert the location of a function to the function between locations. For example, the constructor:

$$\mathsf{int} : \mathsf{primint} \to \mathsf{int}$$

has the corresponding location constructor

$$\mathsf{locap}\ (\mathsf{conloc}\ \mathsf{int}) : (\mathsf{loc}\ \mathsf{primint}) \to \mathsf{loc}\ \mathsf{int}.$$

For example, the proper integer 9 which is a constructed term:

$$\mathsf{int}\ <9> : \mathsf{int}$$

has a corresponding constructed location term of the form

$$\mathsf{locap}\ (\mathsf{conloc\ int})\ (\mathsf{primloc}\ \ {<}9{>})\ \ : \mathsf{loc\ int}.$$

For a constructor of n (zero, one or more) arguments, we need to apply locap n times to the location constructor to create a function between locations. Take our favourite constructor of two arguments bind as an example:

$$\mathsf{bind}\ (\mathsf{ths}\ 1)\ [2] : B\ I\ \mathsf{list\ int}$$

combines two smaller structures of integers ths 1 and the list [2]. The two smaller primitive locations primloc (ths 1) and primloc [2] are combined by the location constructor conloc bind to get:

$$\mathsf{locap}\ (\mathsf{locap}\ (\mathsf{conloc\ bind})\ (\mathsf{primloc}\ (\mathsf{ths}\ 1)))\ (\mathsf{primloc}\ [2]) : \mathsf{loc}(B\ I\ \mathsf{list\ int}).$$

The use of location constructors seems to be long and hard to read but, in many cases, programmers can avoid writing code containing conloc or locap as they can use generic functions that we are going to provide instead. With all the location constructors defined, now we can build generic functions acting on locations. The generic function loc defined in Figure 4.2 creates constructed locations for constructed terms; and creates primitive locations for primitive datum types; it applies conloc otherwise e.g. for functions, commands and locations themselves. It is a simple use of path polymorphism to create constructed locations in a generic way.

```
loc : $X \rightarrow$ loc $X$ =
 | int $t \rightarrow$ locap (conloc int) (primloc $t$)
 | float $t \rightarrow$ locap (conloc float) (primloc $t$)
 | char $t \rightarrow$ locap (conloc char) (primloc $t$)
 | $t_1$ $t_2 \rightarrow$ locap (loc $t_1$) (loc $t_2$)
 | $t \rightarrow$ conloc $t$
```

Figure 4.2: Generic Function loc

For example,

loc (bind (ths 1) (ths 2))

evaluates to

locap (locap (conloc bind) (locap (conloc ths) (locap (conloc int) u_1)))
(locap (conloc ths) (locap (conloc int) u_2))

where u_1 and u_2 are fresh locations holding the primitive integers $<1>$ and $<2>$ respectively.

Here is a bigger example. The concrete representation of the list [9] is tag nm(cons) bind (ths 9) (tag nm(nil) (evr un)), and so

loc [9]

evaluates to

locap (locap (conloc tag) (conloc nm(cons)))
locap (locap (conloc bind) (locap (conloc ths) (locap (conloc int) u)))
(locap (locap (conloc tag) (conloc nm(cons)))(locap (conloc evr) (conloc un)))

where u is a fresh location holding the primitive integer $<9>$.

Similarly, the generic valuation function val is defined in Figure 4.3. It is the inverse of function loc.

```
val : loc X → X  =
 | locap (conloc int) t → int (primval t)
 | locap (conloc float) t → float (primval t)
 | locap (conloc char) t → char (primval t)
 | locap t1 t2 → (val t1) (val t2)
 | conloc t → t
```

Figure 4.3: Generic Function val

For example,

val (locap (locap (conloc bind)(locap (conloc ths) (locap (conloc int) u_1)))
(locap (conloc ths) (locap (conloc int) u_2)))

(where u_1 and u_2 are fresh locations holding the primitive integers $<1>$ and $<2>$ respectively), evaluates to

(bind (ths 1) (ths 2)).

Here is a bigger example,

val (locap (locap (conloc tag) (conloc nm(cons)))
locap (locap (conloc bind) (locap (conloc ths) (locap (conloc int) u)))
(locap (locap (conloc tag) (conloc nm(nil)))(locap (conloc evr) (conloc un))))

(where u is a fresh location holding the primitive integer $<9>$), evaluates to

```
assign : loc $X \to X \to$ comm  =
let rec (assign0 : loc $X \to Y \to$ comm)  =
 | locap (conloc int) $x \to$ ( | int $y \to$ primassign $x$ $y$)
 | locap (conloc float) $x \to$ ( | float $y \to$ primassign $x$ $y$)
 | locap (conloc char) $x \to$ ( | char $y \to$ primassign $x$ $y$)
 | locap $x_0$ $x \to$ ( | $y_0$ $y \to$ assign0 $x_0$ $y_0$; assign0 $x$ $y$)
 | conloc $x \to$ ( | $y \to$ if primequal $x$ $y$ then skip else exn)
in assign0
```

Figure 4.4: Generic Function assign

the list

[9].

A term is called *pure-data* if it is a constructor or an application of two pure-data terms. As primval is the reverse of primloc; val is the reverse of loc.

Lemma 5 *If t is a pure-data term then* val (loc t) *evaluates to t.*

Proof: The proof is by induction on the structure of t. The first case is when t is a constructor c. We know that loc t evaluates to conloc c; and val (loc t) evaluates to c. The second case is when t is the application t_1 t_2. If t is a proper integer, float or character, the proof follows the first three lines of codes in the functions loc and val. If t is of other form, then we can apply the induction hypothesis on t_1 and t_2 and then the proof follows. □

The generic assignment function assign is defined in Figure 4.4. It follows the same basic pattern as the generic functions loc and val but takes two arguments. When a location was created by primloc then the assignment on that location invokes primassign. Otherwise, assign will attempt to match the location constructor with that of the new value. The matching may fail in two ways: first, one of them is an applicative term and the other is not; second, they are distinct constructors. Note that it is not possible to use primassign in such cases, since primassign applies only to primitive locations. The reason for assign to be defined through assign0 of a loose type loc $X \to Y \to$ comm is that the applications $x_0\ x$ and $y_0\ y$ might share the same type while their components x_0 and y_0 have different types. It is the same issue when we define other generic binary functions like equality or plus functions.

When the assignment failure exn occurs, maybe some parts of the location have been assigned to some new value. For example, the assignment assign (loc $[1, 2]$) $[9]$ updates the first element of the list by 9, and then fails when updating the second element. We decide not to back track to recover the old value held in the location. We do acknowledge that might make some programs harder to reason about. One way to avoid the failure is to check the shape before doing the assignment. If the shapes of the location and the new value correspond then the assignment is going to succeed and vice-versa. The generic function for checking the shapes is in Figure 4.5.

As with generic plus or assign, from shapecheck0, one might define another function shapecheck with more restricted type loc $X \to X \to$ bool as

$$(\mathsf{shapecheck} : \mathsf{loc}\ X \to X \to \mathsf{bool}) = \mathsf{shapecheck0}$$

Here is our previous example in Subsection 4.5.1 modified to use generic

```
(shapecheck0 : loc $X \to Y \to$ bool) $u$ $v$ =
match $(u, v)$ with
 | (locap (conloc int) $x$, int $y$) $\to$ true
 | (locap (conloc float) $x$, float $y$) $\to$ true
 | (locap (conloc char) $x$, char $y$) $\to$ true
 | (locap $x_0$ $x$, $y_0$ $y$) $\to$
     shapecheck0 $x_0$ $y_0$ && shapecheck0 $x$ $y$
 | (conloc $x$, $y$) $\to$ primequal $x$ $y$
 | _ $\to$ false
```

Figure 4.5: Shape Checking Function

operations instead of primitive ones:

```
let $x$ = loc [1] in
assign $x$ [2];
assign $x$ [8,9].
```

Now the first assignment is inplace, as desired, but the second assignment will fail, since the structure of nil (coming from the tail of value x currently holding) will not equal to the structure of [9] (coming from the list $[8, 9]$).

4.7 Reference Types

So far, we have a primitive assignment that avoids failure by using fresh memory, and a generic assignment that updates inplace, but may fail if the shapes disagree. These two can be combined on locations of locations (or references) to produce an assignment operation that never fails but is inplace when possible.

```
(ref : $X \to$ ref $X$) $x$ = primloc (loc $x$)
(refval : ref $X \to X$) = val (primval $x$)
(refassign : ref $X \to X \to$ comm) $x$ $y$ =
   match (assign (primval $x$) $y$) with
    | skip $\to$ skip
    | exn $\to$ primassign $x$ (loc $y$).
```

Figure 4.6: Functions on References

The type of *references* of type T is $\mathsf{loc}\ \mathsf{loc}\ T$ also written as

$$\mathsf{ref}\ T$$

Basic functions on references are defined in Figure 4.6. refassign will perform inplace update if possible, but defaults to primitive assignment otherwise.

It is convenient to have some syntactic sugar for operations on locations of locations: let $!x$ denote refval x; and $x := y$ denote refassign x y. Here is our earlier example which is modified to use references:

$$\textbf{let}\ x = \mathsf{ref}\ [1]\ \textbf{in}$$
$$x := [2];$$
$$x := [8, 9].$$

Now the first assignment is inplace and the second assignment is by primitive assignment on the outer location.

4.8 While-Loops and For-Loops

Since we already have general recursion in the language, while-loops and therefore for-loops could be built using tail recursion. The while-loop

$$\textbf{while } t_b \textbf{ do } t_c \textbf{ done}$$

that repeatedly executes command t_c till the boolean t_b becomes false is syntactic sugar $\mathsf{whileloop}\ t_b\ t_c$. Where the recursive function $\mathsf{whileloop}$ is defined as:

$$\begin{aligned}&(\mathsf{whileloop} : \mathsf{bool} \to \mathsf{comm} \to \mathsf{comm})\ t_b\ t_c = \\ &\textbf{let rec } f\ x = \textbf{if } t_b \textbf{ then } (t_c; f\ x) \textbf{ else } \mathsf{skip} \textbf{ in } f\ \mathsf{skip}\end{aligned}$$

Similarly, the for-loop

$$\textbf{for } i = n \textbf{ to } m \textbf{ do } t_c \textbf{ done}$$

that repeatedly executes command t_c for each i in the range of integer from n to m could be seen as syntactic sugar of a function application where the function is built on while-loop:

$$\begin{aligned}&(\mathsf{forloop} : \mathsf{int} \to \mathsf{int} \to (\mathsf{int} \to \mathsf{comm}) \to \mathsf{comm})\ n\ m\ t_c = \\ &\textbf{let } i = \mathsf{ref}\ n \textbf{ in } \mathsf{whileloop}\ (!i \le m)\ (t_c\ !i;\ i := !i + 1)\end{aligned}$$

Obviously, these for- and while-loops can be optimized. Details about that will be discussed later in the section about the implementation.

4.9 Examples

This section uses some examples to illustrate how higher-order functions and pattern-matching can be combined with inplace update, user-control of memory and generic functions to produce short, expressive, space efficient programs.

4.9.1 Bubble-Sort

Bubble-sort [1] works by first sorting the tail and then bubbling the head to its correct position. Here is a purely functional algorithm. First one can define a function that bubbles the head into its correct position.

```
(funbubble : $(X \to X \to \text{bool}) \to X \to \text{list } X \to \text{list } X)$ $g$ $x$ =
| nil → $[x]$
| cons $h$ $t$ → **if** $g$ $x$ $h$ **then** cons $h$ (funbubble $g$ $x$ $t$)
    **else** cons $x$ (cons $h$ $t$)
```

The next part is the main function for sorting:

```
(funbubblesort : $(X \to X \to \text{bool}) \to \text{list } X \to \text{list } X)$ $g$ =
  | nil → nil
  | cons $h$ $t$ → funbubble $g$ $h$ (funbubblesort $g$ $t$)
```

Of couse, we can build funbubble as an application of a generic folding function. However, we want to keep it simple since we want to build its imperative version.

[1] Similar algorithms are called insertion-sort in some other works

```
(swap : loc $X$ → loc $X$ → comm) $x$ $y$ =
  let $t$ = val $x$ in
  assign $x$ (val $y$); assign $y$ $t$

(bubble : ($X$ → $X$ → bool) → loc $X$ → loc list $X$ → comm) $g$ $x$ =
    | conloc nil → skip
    | locap (locap (conloc cons) $h$) $t$ →
          if $g$ (val $x$) (val $h$)
          then swap $x$ $h$; bubble $g$ $h$ $t$
          else skip

(bubblesort : ($X$ → $X$ → bool) → loc list $X$ → comm) $g$ =
    | conloc nil → skip
    | conloc cons $h$ $t$ → bubblesort $g$ $t$; bubble $g$ $h$ $t$
```

Figure 4.7: Imperative Bubble Sort

Suppose the space needed for each list entry is large. By handling locations not their actual values, this algorithm uses space proportional to the square of the list length. The following imperative algorithm bubblesort defined in Figure 4.7 has a similar structure but only uses a constant amount of new memory (when performing swap). The drawback of this program is that the assignments in swap may fail, or be expensive to execute when the structures are large. The solution is to instantiate the polymorphic bubblesort to a type ref Y of locations to get a program of type

$$\mathsf{bubblesort} : (\mathsf{ref}\ Y \to \mathsf{ref}\ Y \to \mathsf{bool}) \to \mathsf{loc\ list\ ref}\ Y \to \mathsf{comm}$$

which can easily be modified to produce a program bubblesortloc in Figure 4.8. bubblesortloc will never fail in assignment but update inplace when

```
(swaploc : ref $X \to$ ref $X \to$ comm) $x$ $y$ =
  let $t$ = !$x$ in
  $x := y$; $y := t$

(bubbleloc : ($X \to X \to$ bool) $\to$ loc ref $X \to$ loc list ref $X \to$ comm) $g$ $x$ =
   | conloc nil $\to$ skip
   | locap (locap (conloc cons) $h$) $t$ $\to$
         if $g$ !(val $x$) !(val $h$)
         then swaploc (val $x$) (val $h$); bubbleloc $g$ $h$ $t$
         else skip

(bubblesortloc : ($X \to X \to$ bool) $\to$ loc list ref $X \to$ comm) $g$ =
   | conloc nil $\to$ skip
   | conloc cons $h$ $t$ $\to$ bubblesortloc $g$ $t$; bubbleloc $g$ $h$ $t$
```

Figure 4.8: Imperative Bubble Sort with Ref

reasonable.

4.9.2 Converge

The function converge defined in Figure 4.9 iterates a function $f : X \to X$ until the result stabilizes, i.e. until some test $t : X \to X \to$ bool applied to the old and new values becomes true. This captures a common situation when modelling the evolution of some system to a steady state. For example, when X is instantiated to the type of real number and f is instantiated to the function cos, and t is the test if two real numbers are close enough (their difference is smaller than some fix number) then converge results in an approximate solution for the equation $\cos x = x$.

The use of explicit locations allows the programmer to indicate that exactly two locations are required at any one time, rather than a unbounded

```
(converge : $(X \to X) \to (X \to X \to \mathsf{bool}) \to X \to X)$ $f\ t\ x =$
let $y$ = ref $x$ in
let $z$ = ref $(f\ x)$ in
while not $(t\ !y\ !z)$ do
  $y := f\ !z$;
  $z := f\ !y$
done;
$!y$
```

Figure 4.9: Converge Function

number. Moreover, if the computation (application of f) does not change the shape of the data then exactly two locations are used for the whole evaluation of the program; e.g. the function cos as described above. A more significant example is by instantiating x to static (fixed shape) object in the physical world, such as using matrixes or three dimensional arrays to represent the heat of plates or solid objects. When the sizes of the matrixes and arrays are large, the guarantee of inplace update is much more needed. Further, assignment will be done inplace if possible, with fresh memory allocated only when necessary. This will yield significant benefits when the shape of the data (y and z) occasional changes through the computation such as operations on graphs and ordered trees. For example, it is common to represent complex dynamical systems using structures built of regions whose behaviours are of approximately equal complexity. If a region is quiet then its representation maintains it shape, and inplace update succeeds. Conversely, if a region is eventful then the shape of its representation is likely to change, and require fresh memory.

4.10 Vector

This section introduces vector types to the system. We do not aim to have efficiently operated data here but just try to show that a vector type can be added to the system. Later in this chapter, we will discuss our vector implementation which is more efficient but also more complicated.

Operationally, a vector is an indexed sequence of locations. We could introduce a vector-like type vector as abstract datatype:

$$\textbf{type}\ \mathsf{vector}\ T = \mathsf{vec}\ \textbf{of}\ (\mathsf{list}\ (\mathsf{loc}\ T)).$$

A data structure of type vector of T can be seen as a (functional) structure of data where each data is a location holding value of type T. Note that the calculus has no constraint on the shapes of different entries. As the shapes might be different, inplace update of one entry by another might fail.

Next, we could provide two functions acting on data of type vector T:

$$(\mathsf{veclen} : \mathsf{vector}\ T \to \mathsf{int}) = \ \mid \mathsf{vec}\ l \to \mathsf{listlen}\ l$$
$$(\mathsf{entry} : \mathsf{vector}\ T \to \mathsf{int} \to \mathsf{loc}\ T) = \ \mid \mathsf{vec}\ l \to \mathsf{listentry}\ l$$

where listlen and listentry are standard operations to calculate the length and to return an entry of a list

$$
\begin{aligned}
&(\mathsf{listlen} : \mathsf{list}\ T \to \mathsf{int}) = \\
&\mid \mathsf{nil} \to 0 \\
&\mid \mathsf{cons}\ h\ t \to 1 + (\mathsf{listlen}\ t) \\
&(\mathsf{listentry} : \mathsf{list}\ T \to \mathsf{int} \to T)\ l\ n = \\
&\textbf{if}\ n < 1\ \textbf{then}\ \mathsf{head}\ l\ \textbf{else}\ \mathsf{listentry}\ (\mathsf{tail}\ l)\ (n-1)
\end{aligned}
$$

veclen v results in the number of components of v. entry v n results in the n^{th} entry of v. Note that entries of vectors are all locations, so they are updatable. Apart from entries access times, the given vector seems to process most standard functionality of vectors.

4.11 Generic Output

4.11.1 Generic Output

This section considers generic output as an imperative feature in the system.

First, consider how it is done in some other languages. ML provides an effective built-in mechanism for output any data, however, one cannot specialize output for some data structures in a generic way. In object-oriented languages such as Java [16], the inheritance mechanism provides programmers with the ability to specialize output for their own classes by overriding the "toString" method. Using monads [67], the "show" function in Haskell [38] also allows the ability to specialize output for each datatype. However, the default case for "show" does not work for newly defined datatypes without mentioning the function "show" in their declaration. There are two main features for the generic output that we are interested in: an effective default display and the ability to specialize the display. ML has the first feature not the second. Java has the second feature. However, if a new class is built from the scratch (it is a child of only the Object class), then the default display ("toString") for that class is minimal. Haskell has both features, however, the use monads does not provide much flexibility in some cases [42].

We will introduce the generic output function in the source code. With the new expresive power from the pattern calculus especially path polymorphism, our generic output function has an effective default display as one of ML and also it has the ability to specialize code of output for any data structure. Moreover, these two features work harmoniously in all levels of nested data structures. One of the significant points of writing the output function in the source code is that it moves a large part of term formatting code out of the compiler.

One might require that the outputting function should hide the tag form representation of abstract datatype, e.g. for outputting the empty list it prints "nil" instead of the concrete representation tag nm(nil) (evr un). We do this in our implementation at the cost of adding one more constant into the system for converting data from the concrete (tag) form to the abstract form. However, we decide not to do it here since the focus of this thesis is not in representation of datatypes.

There are many additional features one could add to the output such as: only printing pairs of brackets in case of possible ambiguity; printing a new line when the number of characters printing reaches a certain number;

alignment of sub-terms in a uniform way - all by adding more auxiliary generic supporting functions. However, we will include only one feature, the bracketing, as an example though other features could be added in similar ways.

4.11.2 Bracketing

When printing an application, strictly speaking, one may always put the argument part under a pair of brackets; however, as a convention in computer science, the pair of brackets are omitted when there is no possible ambiguity. For example, when printing the application of a function f to number 9, instead of printing $f(9)$, it is convention to print just f 9. We need the function:

$$\mathsf{need\_bracket} : X \to \mathsf{bool}$$

that determines if a pair of brackets are needed for a term when it is an argument in an application. In standard cases, the result of the function need_bracket is true when the input is an application and false otherwise, so the last two default cases for the function are:

$$\mid x\ y \to \mathsf{true}$$
$$\mid x \to \mathsf{false}.$$

However, sometimes when we give specialized code for the generic output function, we need to put specialized code for need_bracket too. For example, the specialized printing of list and string does not need brackets to prevent ambiguity, that results in two following lines of specialized code in the func-

```
(need_bracket : $X \to$ bool) =
 | int $x \to$ false
 | float $x \to$ false
 | char $x \to$ false
 | nil $\to$ false
 | cons $x$ $y \to$ false
 | string $x \to$ false
 | pair $x$ $y \to$ false
 | vec $x \to$ false
 | $x$ $y \to$ true
 | $x \to$ false
```

Figure 4.10: Need-Bracket Function

tion need_bracket:

```
| cons $x$ $y \to$ false
| string $x \to$ false.
```

The complete code for the function need_bracket is defined in Figure 4.10.

We now can create an auxiliary function print_bk, together with the function output, they form a mutually recursive pair of functions. The function print_bk prints a pair of brackets when needed while performing printing of the input:

```
(print_bk : $X \to$ comm) $t$ =
**if** need_bracket $t$ **then**
  output_basic <'('>;   output $t$;   output_basic <')'>
**else**
  output_basic <' '>;   output $t$.
```

4.11.3 Supporting Functions for Output

The output function written in Figure 4.11 calls two supporting functions. The first one is an auxiliary function named listiter, it is a simple application fold-left on list:

```
(listiter : (X → comm) → list X → comm) f =
| nil → skip
| cons h t → (f h); listiter f t.
```

The second one, the built-in operator output_basic is introduced earlier in this chapter as an imperative feature. The function provides a basic built-in mechanism for displaying constructors (especially primitive datum constructors). For example output_basic $<9>$ and output_basic $<'a'>$ will print integer 9 and character a on the output device respectively; output_basic un will print the string "un" on the output device (without the quotation marks).

4.11.4 Output Function

This section discusses the generic output function. Our implementation of output adopts the following typical syntax for displaying pairs, lists, strings and vectors, for example:

the list of three integers 1, 2 and 3 is to be output as $[1, 2, 3]$;

the pair of integer 0 and float 9.9 is to be output as $(0, 9.9)$;

the string representing the word Tony is to be output as "Tony";

the vector of two integers 0 and 9 is to be output as $\{0, 9\}$.

```
(output : $X \rightarrow$ comm) =
    | int $x \rightarrow$ output_basic $x$
    | float $x \rightarrow$ output_basic $x$
    | char $x \rightarrow$ output_basic $x$
    | pair $x$ $y \rightarrow$
        output_basic <'('>;
        output $x$;
        output_basic <','>;
        output $y$;
        output_basic <')'>;
    | nil $\rightarrow$ output_basic <'['>;   output_basic <']'>
    | cons $h$ $t \rightarrow$
        output_basic <'['>;
        output $h$;
        listiter (fun $v \rightarrow$ (output_basic <','>; output $v$)) $t$;
        output_basic <']'>;
    | string $x \rightarrow$
        output_basic <'"'>;
        listiter output_basic $x$;
        output_basic <'"'>
    | vec $x \rightarrow$
        output_basic <'{'>;
        let $l$ = veclen (vec $x$) in
        if ($l = 0$) then
          skip
        else
          output (primval (entry (vec $x$) 0));
          for $i = 1$ to ($l - 1$) do
            output_basic <','>;
            output (primval (entry (vec $x$) $i$))
          done;
        output_basic <'}'>;
    | $x$ $y \rightarrow$ output $x$;   print_bk $y$
    | $x \rightarrow$ output_basic $x$
```

Figure 4.11: Generic Output Function

These displays are different from the results of the default case of function output. New pattern-matching to constructors: nil and cons; pair; string; and vec make the above differences in the cases of corresponding structures of list, pair, string and vector. The output code is in Figure 4.11. Note that the user can augment or change the output function to specialize the display of some particular data structures. In order to do that, the user just needs to add new cases on constructors for creating the data structures as the first pattern-matches in the output function. Note that complex numbers have been defined in Chapter 2 as:

type Complex = Complex **of** primfloat **and** primfloat.

Similar to the special rule for multiplication of complex numbers. we can put the new specialized code for displaying complex numbers:

```
| Complex x y →
  output_basic x;
  output_basic <'+'>;
  output_basic <'i'>;
  output_basic y.
```

4.12 Implementation

As stated in the introduction chapter, all material has been implemented. All examples have been coded and tested; and they produced expected results. Codes are available on request.

We [2] could have implemented the exact system as presented above. However, in some points in our implementation, we make some derivations from the theory, mostly for optimization purposes.

The theory in the previous sections introduces sequential composition as a form of an application of an extension.

$$(\mathsf{seq} : \mathsf{comm} \to X \to X) = \ \mid \mathsf{skip} \to (\ \mid y \to y)\ .$$

However, the implementation adds seq as another constant of the language with an optimized evaluation rule. We could modify the theory by adding a new evaluation rule for seq to meet the implementation but we choose not to do so for the simplicity of the theory.

Similar derivations are with the implementation of while-loop and for-loop. Our implementation adds them as new constants of the system with optimized evaluation rules.

The implementation for vectors is more delicate. Note that the definition of vector in the theory (in Section 4.10) results in a datatype with linear accessing time. We aim to have vectors with constant accessing time. For efficiency reason, instead of using the type list (loc X) as in the theory, we introduce a new type constant:

$$\mathsf{primvector}.$$

Each type X has an associated type primvector X. We also introduce a new

[2]For detail about my involvement, see Subsection 1.2.1 in Chapter 1

class of terms of primitive vectors of those primitive vector types. A primitive vector of type primvector X contains a collection of values of type X in a way that supports constant accessing time. [3]

To pattern-match on vectors, as the same way we deal with integers, vectors are defined based on the primitive type as:

$$\textbf{type}\ \mathsf{vector}\ X = \mathsf{vec}\ \textbf{of}\ (\mathsf{primvector}\ X).$$

For creating a new primitive vector of given length whose components are all initialized to the same given value, we introduce a new built-in term:

$$\mathsf{locvec} : \mathsf{int} \to X \to \mathsf{primvector}\ X.$$

A term of the form locvec n x creates n new distinct locations (in the environment store) all initialized to the value x, and a new structure (primitive vector) holding these locations. We also introduce vecnil as built-in term for vector of length 0.

We also introduce two new constants associated with primitive vectors:

$$\begin{aligned} \mathsf{primveclen} &: \mathsf{primvector}\ X \to \mathsf{int} \\ \mathsf{primentry} &: \mathsf{primvector}\ X \to \mathsf{int} \to \mathsf{loc}\ X. \end{aligned}$$

Given a primitive vector v and an integer i, the term primveclen v represents the length of v, and the term primentry i v represents the i^{th} component of v. Out of bound vector error, i.e. when i is negative or greater than or equal

[3]We use arrays in the host language (OCAML) to ensure that property.

to the length of v, will be caught as a runtime exception. The key point of getting all new primitives for vectors is that now accessing time for entries is constant.

Operations on abstract vectors can be derived from the corresponding operations on primitive types by simple pattern-matching:

$$\begin{aligned}
&(\mathsf{veclen} : \mathsf{vector}\ X \to \mathsf{int}) = \ \mid \mathsf{vec}\ v \to \mathsf{primveclen}\ v \\
&(\mathsf{entry} : \mathsf{vector}\ X \to \mathsf{int} \to \mathsf{loc}\ X) = \ \mid \mathsf{vec}\ v \to (\\
&\qquad \mid \mathsf{int}\ n \to \mathsf{primentry}\ v\ (\mathsf{int}\ n))
\end{aligned}$$

We also introduce syntactic sugar for creating vectors. A new vector can be created by specifying all of its components. An expression starting and ending with { and } respectively and enclosing the body of a non-empty list of terms of type X separating by commas is a new vector of X. For example, the expression $\{1.1, 2.3\}$ represents a new vector containing two floats. Note that the term represented by { } is a new vector of zero length, vec vecnil.

In our implementation, the outputting function hides the tag form representation of an abstract datatype. e.g. for outputting the empty list it prints "nil" instead of the concrete representation tag nm(nil) (evr un); at the cost of adding one more constant into the system for converting data from the concrete tag form representation to the abstract representation.

Our implementation also includes the standard type inference mechanism. It is based on the type inference in the pattern calculus. The type inference is working for all examples we have tried (including all the examples presented in this thesis).

4.13 Limitations

Before concluding this chapter, we discuss some limitations of the system some of which can be seen as motivation for the building the uniform system (in the next chapter). Although the idea of location constructors is similar to that of normal constructors, the actual uses of location constructors are long and complicated. As the number of constructors is doubled by adding location constructors, there is a possibility that in some parts of coding, the programmers have to double the amount of pattern-matches in dealing with constructors and location constructors.

Another limitation is the separation of inplace and out-place updates, that might lead to inefficiency when assigning a large data structure and at a later point discovering that the shapes are not matching and having to reassign everything out-place. The decision of not back-tracking when inplace update fails might make some programs hard to reason about; especially when coupling with aliasing which by itself is a complicated topic [13, 55]. The following example tries to illustrate the point. In the expression:

$$\begin{array}{l}\textbf{let}\ x = \mathsf{loc}\ v\ \textbf{in}\\ \textbf{let}\ y = \mathsf{primloc}\ x\ \textbf{in}\\ y := u;\quad \mathsf{val}\ x\end{array}$$

the return value $\mathsf{val}\ x$ depends on the internal mechanism with which the assignment $y := u$ is done. If u and v are of the same shape then the assignment is by inplace and hence the return value is the same as v. If u and v are of different shapes then the attempt to assign inplace fails, the attempt changes some parts of the data held in x to be the corresponding parts in v. When performing the out-place assignment, the value held in x does not change, and hence the result val x has mixed part between u and v. However, by using ref, refval and refassign but not the primitive ones we do not have access to the inner layer (ref creates two layers of location) and hence we can avoid the problem.

The confusion is a part of the price to be paid when having the purely functional types separated from the location types. To use imperative feature, the users usually have to know the different between functional and imperative types; and have to convert data from one form to another using operations such as loc and val. In the next chapter, a different approach that unifies the functional and imperative types will help to overcome some of those limitations.

4.14 Conclusions

Based on the pattern calculus as the theory for a purely functional language, this chapter is about the extension of the calculus with imperative features. One of the objectives is to add more expressive power to the system with the minimum effect on the purely functional features. The clear separation between functional types and imperative types allows choices for which style is to be used. Based on path polymorphism that is to build generic functions in the pattern calculus, this chapter shows that imperative operations can be used generically for creating, reading from and writing to constructed locations. The advantage of this approach is that it naturally supports inplace update, something that is hard to achieve in other languages supporting recursive types. By adding a layer of indirection, one can wrap inplace update within a safe form of assignment on reference types.

The expressive power of the approach is shown through some representative examples. Bubble-sort shows how the functional programming style, with its pattern-matching and recursion can be used to define efficient imperative code. The converge program nicely illustrates the value of sharing control between the programmer and the system: the programmer specifies how many data structures are required while the system determines when fresh storage is required. The examples also show how the power of the generic programming style combines naturally with the imperative features to provide flexible programming on large data structures.

The ideas and examples in this chapter show that the pattern calculus is able to combine the functional and imperative programming styles within a single, simple calculus. The added features of while-loops and for-loops, vector types and the generic output function is evidence for the orthogonality of the system. The addition of loops and vector types shows the potential of optimizing the system without affecting its soundness. The addition of a generic output function also shows that exploiting the expressive power of the system can help to reduce the complexity of the compiler.

Chapter 5

Mutable Data

5.1 Introduction

This chapter, like the previous chapter, is about extending the pattern calculus with imperative features. The theory is (again) implemented as a version of FISh2 with imperative features. In the previous chapter, imperative features are added on top of a functional system, the pattern calculus. In particular, there is a new class of location terms (updatable locations) whose types are distinct ones of their value. In this chapter, the type of a location is identical to the type of the value held in the location. All data are potentially updatable and so must be stored in mutable locations. From the user's viewpoint, all data are handled through implicit references; we provide a cloning constant for copying the actual data.

The previous chapter introduces location constructors with the use of conloc and locap. Although the theory is simple, the actual uses of location constructors are long and potentially confusing. There is a solid reason for the complexity, the uses of conloc and locap maintain the type separation between references and their values. This chapter will try to shift the task to the calculus so that the users do not need to handle referencing and dereferencing explicitly so that constructed locations are easier to describe and use than in the previous chapter.

In spite of the above differences, there are still many common points between the two systems in the previous and this chapter. Both systems use value polymorphism as described in Chapter 3 in dealing with polymorphism and updatable locations. Both systems use the power of generic functions from the pattern calculus to build a generic assignment function which provides a flexible and efficient mechanism for memory management. Many other aspects of the two systems such as generic output, vector, while- and for-loops are very similar, in both theory and implementation. Hence, we omit them in this chapter.

This chapter will explore some of the different choices in building such a system to effectively setting on one. A summary of benefits and weaknesses will be made at the conclusion of this chapter. The goal here is to explore many possible interesting examples in a well-typed system. We do not claim that the system is the best nor that it contains all of the desired features.

5.2 Types and Terms

The types and the terms of this system are built on those of the pattern calculus. Like the previous chapter, we use value polymorphism to accommodate both updatable locations and polymorphic functions, so it is sufficient to introduce some imperative constants.

Type	comm
Term	skip : comm
	primassign : $\forall X.X \to X \to$ comm
	clone : $\forall X.X \to X$
	output_basic : $\forall X.X \to$ comm

First, we introduce the command type comm with the associate constructor skip of type comm. skip is a constructor so that one can pattern-match on it. The arity of skip is defined specially to be positive infinity to ensure that skip will never be stored [1]. Second, we introduce primitive assignment primassign with given type scheme $\forall X.X \to X \to$ comm. Next, we introduce the cloning operation clone with given type scheme $\forall X.X \to X$. Finally, we introduce a primitive printing operation output_basic exactly as in the previous chapter for printing constructors.

Comparing to the previous chapter, there is no explicit location type here and hence no operations to create new location or getting value from a location. The primitive assignment can take the first input of any type. So any values that might be assigned have to be locations (references) holding some terms. More details about primassign and clone will be discussed later when the store has been formally defined.

Sequential composition can be defined exactly the same as in the previous

[1]It is a small choice in designing the system. One might choose to store skip which results in a little different system.

chapter:

$$(\mathsf{seq} : \mathsf{comm} \to X \to X) = \ | \ \mathsf{skip} \to (\ | \ y \to y) \ .$$

We also may use the usual syntactic sugar $x; y$ for $\mathsf{seq}\ x\ y$.

5.3 Evaluation

In the previous chapters, the set of possible results of an evaluation (value) is the same as the set of terms which can be stored. However, in this chapter, they are two different sets. In particular, (explicit) functions will be values that are not stored; and the content of store variables will not be returned as the results of evaluation (but the store variable will be).

First, we define what can be a result of an evaluation. A *value*, meta-variable v, is given by:

$$v ::= x \mid \mathbf{at}\ p\ \mathbf{use}\ s\ \mathbf{else}\ t \mid b\ v_1\ \ldots\ v_k\ \{k < \text{arity}(b)\}.$$

That is, a value is either: a term variable; an extension; or a partially-applied constant whose arguments are all values. Some examples of values are: x; $\lambda x.x$; cons; and int. Some examples of non-value terms are: nil; $x\ 9$; and $\mathsf{int} < 9 >$ (the proper integer 9).

Note that all values are either term variables or explicit functions (including the skip constructor and terms headed by exn). Fully-applied constructors are to be stored in locations. Fully-applied operators require their own evaluation rules.

Next, we define what can be stored in a location. A term is called *storable* (or *data*) if it is a fully-applied constructor where all the arguments are values. Some examples of storable terms are: nil; 9; and cons $\lambda x.x\ y$.

Axiom 6 *A well-typed fully-applied constructor has data-form type.*

Note that a data-form type is defined in Chapter 2 to be a type constant or an application of two types. From the axiom, we know that all data has data-form types.

In this chapter, the distinction between a location and the actual term held in the location are handled inside the calculus. So the definitions of values and data capture some major design choices for the system. The definition of values determines the possible results of evaluations. The definition of data determines which terms can be updated. Comparing with the previous chapter, the choices are also there, but for the users of the system to choose. For example, in the previous chapter, there are two choices for constructing a "larger" location: by creating one location out of one large term (using primloc); or by combining smaller locations (using conloc and locap). The flexibility also causes the complexity (in the syntax and reasoning) in the uses. One of the changes in this chapter is that we make a fixed choice (inside the calculus) when defining storable terms.

In the previous chapter, we introduce locations in two steps. First, locations created by primloc are atomic or unstructured. Second, constructed locations are built on atomic ones and location constructors to support pattern-matching. In this chapter, the choices of atomic or constructed locations come in the design of the system. The decision that a fully-applied constructor is storable only when its arguments are values (not storable terms) implies that "larger" locations are always constructed out of "smaller" ones. In a typical larger location, the content of the location is the application of a constructor to other smaller locations. This choice of constructed locations allows partial update of a larger location which will be exploited later in the chapter.

A key point in designing such a system is the right balance between values and storable terms. Our choice here for not storing function is similar to the Object-Oriented (OO) approach. An extension (a function, a method) such as $\lambda x.x$ by itself is not data (not a updatable term, not an object). However, by wrapping an extension inside an constructed term such as $\mathsf{ths}\ \lambda x.x$, we have an storable term or an object in the OO view. One alternative is to define values to be variables only and to expand the set of storable terms with extension.

The definitions of: a *store* (meta-variable Σ); *store variable* (meta-variable u); and an *evaluation context* (Σ, t) are exactly as in the previous chapter (see Section 4.3. Again, we decide that all store variables have monomorphic (trivial) type scheme. We use $\mathsf{dom}\ \Sigma$ for the domain of store Σ. In the context of a given store Σ, for a term t we define $!t$ to be $\Sigma(t)$ if t is a term variable in the domain of Σ and t otherwise. This ! mechanism plays the role of the $\mathsf{primval}$ constant in the previous chapter.

Evaluation rules are expressed the same way as in the previous chapter using judgements of the form

$$(\Sigma, t) \Rightarrow (\Sigma', v)$$

where (Σ, t) and (Σ', v) are evaluation contexts.

Rule 1 $$\frac{(\Sigma, t[\mathbf{fix}\ (x,t)/x]) \Rightarrow (\Sigma', v)}{(\Sigma, \mathbf{fix}\ (x,t)) \Rightarrow (\Sigma', v)}$$

Rule 2 $$\frac{(\Sigma, s) \Rightarrow (\Sigma', v_0) \quad (\Sigma', t[v_0/x]) \Rightarrow (\Sigma'', v)}{(\Sigma, \mathbf{let}\ x = s\ \mathbf{in}\ t) \Rightarrow (\Sigma'', v)}$$

Rule 3 $$\frac{(\Sigma, s[v_2/x]) \Rightarrow (\Sigma', v)}{(\Sigma, (\mathbf{at}\ x\ \mathbf{use}\ s\ \mathbf{else}\ t)\ v_2) \Rightarrow (\Sigma', v)}$$

Rule 4 $$\frac{(\Sigma, s) \Rightarrow (\Sigma', v)}{(\Sigma, (\mathbf{at}\ c\ \mathbf{use}\ s\ \mathbf{else}\ t)\ v_2) \Rightarrow (\Sigma', v)}\ !v_2 = c$$

Rule 5

$$\frac{(\Sigma, (\mathbf{at}\ p_1\ \mathbf{and}\ p_2\ \mathbf{use}\ s\ \mathbf{else}\ t)\ t_1\ t_2) \Rightarrow (\Sigma', v)}{(\Sigma, (\mathbf{at}\ p_1\ p_2\ \mathbf{use}\ s\ \mathbf{else}\ t)\ v_2) \Rightarrow (\Sigma', v)}\ \begin{array}{l} !v_2 = t_1\ t_2 \\ t_1 \text{ is a constructed term}\end{array}$$

Rule 6 $$\frac{(\Sigma, t\ v_2) \Rightarrow (\Sigma', v)}{(\Sigma, (\mathbf{at}\ p\ \mathbf{use}\ s\ \mathbf{else}\ t)\ v_2) \Rightarrow (\Sigma', v)}$$

Rule 7

$$\frac{}{(\Sigma, \mathsf{primequal}\ v_1\ v_2) \Rightarrow (\Sigma, u \mapsto \mathsf{true}, u)}\ \begin{array}{l} !v_1, !v_2 \text{ are the same constructor} \\ u \text{ fresh}\end{array}$$

Rule 8 $$\frac{}{(\Sigma, \mathsf{primequal}\ v_1\ v_2) \Rightarrow (\Sigma, u \mapsto \mathsf{false}, u)}\ u \text{ fresh}$$

Rule 9

$$\frac{}{(\Sigma, \mathsf{output\_basic}\ v) \Rightarrow (\Sigma, \mathsf{skip})}\ \text{output } !v \text{ if } !v \text{ is a constructor}$$

Rule 10 $$\frac{}{(\Sigma, v\ v_1) \Rightarrow (\Sigma, v_1)}\ \begin{array}{l} v \text{ headed by an operator} \\ v_1 \text{ headed by } \mathsf{exn}\end{array}$$

Figure 5.1: Evaluation Rules with Mutable Data 1

The evaluation rules are defined in two parts: Figure 5.1 and Figure 5.2. Most of the rules are the same or very similar to the ones in the previous chapter. The rules in the first figure are based on the reduction rules (discussed in the review of the pattern calculus). The rules in the second figure are mainly about the imperative features. The main different from the previous chapter is the rules for primassign. Evaluation of assignment on terms that cannot become data such as functions results in exception. That is a consequence of our choice of storable terms. The rules for clone acts in usual way, evaluation of clone will make a fresh copy (clone) of a store variable. If the argument of clone is any value other than a store variable, the result of evaluation is an exception.

Rule 11

$$\frac{(\Sigma, t) \Rightarrow (\Sigma', u) \quad (\Sigma', t_2) \Rightarrow (\Sigma'', u_2)}{(\Sigma, \mathsf{primassign}\ t\ t_2) \Rightarrow (\Sigma'', u \mapsto \Sigma''(u_2), \mathsf{skip})} \quad \begin{array}{l} u \in \mathsf{dom}(\Sigma') \\ u_2 \in \mathsf{dom}(\Sigma'') \end{array}$$

Rule 12

$$\frac{(\Sigma, t) \Rightarrow (\Sigma', v) \quad (\Sigma', t_2) \Rightarrow (\Sigma'', v_2)}{(\Sigma, \mathsf{primassign}\ t\ t_2) \Rightarrow (\Sigma'', \mathsf{exn})}$$

Rule 13

$$\frac{}{(\Sigma, \mathsf{clone}\ u_1) \Rightarrow (\Sigma, u \mapsto \Sigma(u_1), u)} \quad \begin{array}{l} u_1 \in \mathsf{dom}\ \Sigma \\ u \text{ fresh} \end{array}$$

Rule 14

$$\frac{}{(\Sigma, \mathsf{clone}\ v) \Rightarrow (\Sigma, \mathsf{exn})}$$

Rule 15

$$\frac{}{(\Sigma, v) \Rightarrow (\Sigma, v)}$$

Rule 16

$$\frac{}{(\Sigma, d\ u_1 \cdots\ u_n) \Rightarrow (\Sigma, u \mapsto d', u)} \quad \begin{array}{l} \Sigma(u_i) = d_i \\ d\ d_1 \cdots\ d_n = d' \\ u \text{ fresh} \end{array}$$

Rule 17

$$\frac{}{(\Sigma, c\ v_1 \ldots\ v_k) \Rightarrow (\Sigma, u \mapsto c\ v_1 \ldots\ v_k, u)} \quad \begin{array}{l} c\ v_1 \ldots\ v_k \text{ is storable} \\ u \text{ fresh} \end{array}$$

Rule 18

$$\frac{(\Sigma, t_1) \Rightarrow (\Sigma', v_1) \quad (\Sigma', t_2) \Rightarrow (\Sigma'', v_2) \quad (\Sigma'', v_1\ v_2) \Rightarrow (\Sigma''', v)}{(\Sigma, t_1\ t_2) \Rightarrow (\Sigma''', v)} \quad \begin{array}{r} t_1, t_2 \text{ are not} \\ \text{both values} \end{array}$$

Figure 5.2: Evaluation Rules with Mutable Data 2

The rest of this section establishes some standard properties of evaluation. Like the previous chapter, we define a new property about the relation between the typing of the store variables and of their values. The judgement $\Delta; \Gamma \vdash_\Sigma \Sigma$ asserts that the store Σ is *well-typed* in context $\Delta; \Gamma$. Suppose Σ is $\{u_1 \mapsto v_1, \ldots, u_n \mapsto v_n\}$ then $\Delta; \Gamma \vdash_\Sigma \Sigma$ if and only if $\Sigma; \Gamma$ is well-formed and there are types T_i such that $\Gamma(u_i) = T_i$ and $\Delta; \Gamma \vdash v_i : T_i$.

In other words, in the type context $\Delta; \Gamma$, a store Σ is called well-typed if the domain of Σ is a subset of the domain of Γ and each store variable in Γ must be of the type corresponding to the type of its value in Σ.

As in the previous chapter, the type context has to expand during the evaluation to cope with the new store variables. In particular, when a term is one of the four cases: a fully-applied primequal; a storable term; a fully applied datum operation; or a cloning of a store variable, the evaluation of the term results in a new store variable u which must be added to the context as well as the store.

The evaluation rules for all the four cases are of the form

$$\frac{}{\ldots \Rightarrow (\Sigma, u \mapsto t_I, u)}$$

where t_I is the initial term. Note that t_I is storable. We can extend the type context with a new variable u whose type (scheme) is the type of t_I.

$$\frac{\Delta; \Gamma \vdash_\Sigma \Sigma \quad \Delta; \Gamma \vdash t_I : T}{\Delta; \Gamma, u : T \vdash_\Sigma \Sigma, u \mapsto t_I}\; u \text{ fresh}$$

By that choice of extending the type context, if the store is well-typed before, then clearly it remains well-typed after the evaluation.

As in the previous chapter, we introduce a new form of judgement for typing an evaluation context. The judgement $\Delta;\Gamma \vdash_i (\Sigma, t) : T$ asserts that the evaluation context (Σ, t) is well-typed and t has type T in the context of $\Delta;\Gamma$ and the store Σ.

The theorem about evaluation preserving typing is exactly the same as Theorem 2 in the previous chapter.

Theorem 7 *Evaluation preserves typing. That is, if there are derivations $\Delta;\Gamma \vdash_i (\Sigma, t) : T$ and $(\Sigma, t) \Rightarrow (\Sigma', v)$ then there is a context Γ_1 such that $\Delta;\Gamma,\Gamma_1 \vdash_i (\Sigma', v) : T$.*

Proof: The proof is done by induction on the structure of evaluation based on the evaluation rules in Figure 5.1 and Figure 5.2. For all of the cases, the proofs are identical or very similar to the ones of Theorem 2 in the previous chapter. Examples of the differences are the cases for Rules 11,12,13, and 16.

Rule 11

$$\frac{(\Sigma, t) \Rightarrow (\Sigma', u) \quad (\Sigma', t_2) \Rightarrow (\Sigma'', u_2)}{(\Sigma, \mathsf{primassign}\ t\ t_2) \Rightarrow (\Sigma'', u \mapsto \Sigma''(u_2), \mathsf{skip})} \quad \begin{array}{l} u \in \mathsf{dom}(\Sigma') \\ u_2 \in \mathsf{dom}(\Sigma'') \end{array}$$

From the assumption $\Delta;\Gamma \vdash_i (\Sigma, \mathsf{primassign}\ t\ t_2) : T$ it follows that $\Delta;\Gamma \vdash_\Sigma \Sigma$ and there is some type T_1 such that $\Delta;\Gamma \vdash_i (\Sigma, t) : T_1$. Applying the induction hypothesis on the evaluation of (Σ, t) we know there is Γ_1 such that $\Delta;\Gamma,\Gamma_1 \vdash_i (\Sigma', u) : T_1$. Applying the induction hypothesis on the evaluation of (Σ', t_2) we know there is Γ_2 such that $\Delta;\Gamma,\Gamma_1,\Gamma_2 \vdash_i (\Sigma'', u_2) : T_1$. From the definition of a well-typed store we know $\Delta;\Gamma,\Gamma_1,\Gamma_2 \vdash \Sigma''(u_2) : T_1$ and hence $\Delta;\Gamma,\Gamma_1,\Gamma_2 \vdash_\Sigma \Sigma'', u \mapsto$

$\Sigma''(u_2)$. Moreover, from the typing of $\mathsf{primassign}$ we know that T is comm and we have $\Delta;\Gamma,\Gamma_1,\Gamma_2 \vdash_i (\Sigma'', u \mapsto \Sigma''(u_2), \mathsf{skip}) : T$.

Rule 12

$$\frac{(\Sigma, t) \Rightarrow (\Sigma', v) \quad (\Sigma', t_2) \Rightarrow (\Sigma'', v_2)}{(\Sigma, \mathsf{primassign}\ t\ t_2) \Rightarrow (\Sigma'', \mathsf{exn})}$$

Since exn can take the type comm, the proof for this case follows the one of Rule 11.

Rule 13

$$\frac{}{(\Sigma, \mathsf{clone}\ u_1) \Rightarrow (\Sigma, u \mapsto \Sigma(u_1), u)} \quad \begin{array}{l} u_1 \in \mathsf{dom}\ \Sigma \\ u \text{ fresh} \end{array}$$

Since the store is well-typed before the evaluation, the store remains well-typed after cloning a store variable by choosing the same type for u as one of u_1. Moreover, from the type of clone we know that $\mathsf{clone}\ u_1$ and u shared the same type. The proof for this case follows.

Rule 16

$$\frac{}{(\Sigma, d\ u_1 \cdots\ u_n) \Rightarrow (\Sigma, u \mapsto d', u)} \quad \begin{array}{l} \Sigma(u_i) = d_i \\ d\ d_1 \cdots\ d_n = d' \\ u \text{ fresh} \end{array}$$

From the assumption $\Delta;\Gamma \vdash_i (\Sigma, d\ u_1 \cdots\ u_n) : T$ it follows that $\Delta;\Gamma \vdash_\Sigma \Sigma$. Hence d_i has the same type as u_i for all i. So we know that $d\ d_1 \cdots\ d_n$ (or d') has type T. Moreover the type (scheme) of the new store variable u is chosen to be the type of d'. We can conclude that $\Delta;\Gamma \vdash_i (\Sigma, u \mapsto d', u) : T$.

□

The definitions of closedness are exactly as in the previous chapter. A store Σ is called *closed* if for every variable u in the domain of Σ, all free variables of $\Sigma(u)$ are in the domain of the store Σ. The evaluation context (Σ, t) is called *closed* if Σ is closed and all free variables of the term t are in the domain of Σ. As in the previous chapter, we have the lemma:

Lemma 8 *Evaluation preserves closedness. That is, suppose there is an evaluation* $(\Sigma, t) \Rightarrow (\Sigma', v)$ *and* (Σ, t) *is closed then* (Σ', v) *is also closed.*

Proof: Again, the proof is by induction on the structure of evaluation. We do a case analysis on the evaluation rules in Figure 5.1 and Figure 5.2. For all of the cases, the proofs are identical or very similar to the ones in the previous chapter. $\Box$

As in the previous chapter, combining this above lemma with the previous theorem we can claim that evaluation preserves well-typedness and closedness. A closed and well-typed term is called a *program.* We have the same theorem as in the previous chapter discussing programs evaluation.

Theorem 9 *Evaluation of programs never gets stuck. That is, if we have a derivation* $\Delta; \Gamma \vdash_i (\Sigma, t) : T$ *and* (Σ, t) *is closed then there is an evaluation rule where the left hand side of the conclusion matches with* (Σ, t).

Proof: The proof is similar to the one of Theorem 4 in the previous chapter. The proof is by case analysis on the structure of t, and it is mostly based on the definition of values and the evaluation rules in Figure 5.1 and Figure 5.2.

When t is a variable or an extension then t is a value. We could always apply Rule 15 which states that any value evaluates to itself.

When t is a fix point construction, then the first rule applies.

When t is a let-expression, then the second rule applies.

When t is a constant then t is either storable hence Rule 17 applies; or partially-applied (a value) which we can apply the rule for values too.

The only case left is when t is an application. As one could try and apply the last rule when at least one of the parts of the application is not a value, without loss of generality, now we can assume that t has the form $v_0\ v_1$ where v_0 and v_1 are values. Now consider the cases for the choices of v_0.

If v_0 is a variable, the closed condition of the evaluation context forces v_0 to be a variable in the domain of the store. The well-typedness of the store forces v_0 to be of data-form type (the same type as some data stored in v_0). However, the function part of an application (always of function type) can never be of data-form type, therefore this case never happens;

If v_0 is an extension, then Rule 6 applies.

If v_0 is headed by a constant b, then t is also headed by b. By the well-typedness of t and the definition of values (as v_0,v_1 are values), there are three cases of such a term t. Case one, t is partially-applied and hence a value, we can apply the rule which states that any value evaluates to itself. Case two, t is a fully-applied operator whose arguments are all values, we can apply one of the Rules 8,9,11,12,14 and 16. Case three, t is a fully-applied constructor

```
(assign : $X \to X \to$ comm) $=$
let rec (assign0 : $X \to Y \to$ comm) $=$
 | int $x \to$ ( | int $y \to$ primassign $x$ $y$)
 | float $x \to$ ( | float $y \to$ primassign $x$ $y$)
 | char $x \to$ ( | char $y \to$ primassign $x$ $y$)
 | $x_0$ $x \to$ ( | $y_0$ $y \to$ assign0 $x_0$ $y_0$; assign0 $x$ $y$)
 | $x \to$ ( | $y \to$ if primequal $x$ $y$ then skip else exn)
in assign0
```

Figure 5.3: Generic Function assign using Mutable Data

whose arguments are all values, then t is storable, we can apply the Rule 17.

□

5.4 Examples

This section will discuss some typical examples showing the expressive power of the system which is (almost) equivalent to the one of previous chapter. The major difference from the examples in the previous chapter is the loss of conloc and locap.

5.4.1 Generic Assignment Function

The generic assignment in Figure 5.3 has the same structure to the generic assign in Figure 4.4 in the previous chapter. The only different is that the function in this chapter contains no conloc nor locap.

This function has the same strength and weakness as the one in the

```
(refassign : ref $X \rightarrow X \rightarrow$ comm) $x\ y =$
  match (assign $x\ y$) with
   | skip $\rightarrow$ skip
   | exn $\rightarrow$ primassign $x\ y$.
```

Figure 5.4: Generic Function refassign using Mutable Data

previous chapter. Most importantly, the generic assignment still tries to do the inplace update if the structures of the new data and the old location match, and fails (return an exception) otherwise. Another example for the similarity, when an assignment failure occurs, parts of the location might have been assigned to new values. We can avoid the problem using the same method as in the previous chapter: checking the shapes before assignment by a function similar to shapecheck function in Figure 4.5.

In the previous chapter, we have a mixture of inplace and primitive assignment by using two levels of location (see Section 4.7). We can do the same thing here in a simpler form.

As in the previous chapter, refassign in Figure 5.4 will perform inplace update if possible, but defaults to primitive assignment otherwise.

5.4.2 Infinite Lists

Similar to the self linked list in the previous chapter, here, an infinite list can be represented using self references. The program

```
let $x = [\,]$ in
primassign $x$ (cons 9 $x$); $x$
```

terminates. The result of evaluation is a store variable u where the store is $\{u_2 \mapsto <9>, u_1 \mapsto \mathsf{int}\ u_1, u \mapsto \mathsf{cons}\ u_1\ u\}$. So u is a recursive location and it can also be seen as a list holding infinite number of integers 9.

Comparing to the linked-lists (in Subsection 4.5.2) in the previous chapter, we do not have to define a new datatype for lists with locations. However, updating a list involving itself without creating a recursive location can be done using clone. For example, the program

$$\begin{array}{l}\textbf{let } x = [0] \textbf{ in}\\ \mathsf{primassign}\ x\ (\mathsf{cons}\ 9\ (\mathsf{clone}\ x)); x\end{array}$$

results in the list $[9, 0]$.

5.4.3 Graphs

We define a simple polymorphic directed graph holding two sorts of data: one for the nodes, the other for the edges.

$$\textbf{type}\ \mathsf{node}\ (X, Y) = \mathsf{node}\ \textbf{of}\ X\ \textbf{and}\ (\mathsf{list}\ ((\mathsf{node}\ (X, Y)) * Y)).$$

A node (of type node (X, Y)) can be created by applying the constructor node to some term of type X and a list of edges from that node, each edge is a pair of a node (of (X, Y)) and some term of type Y.

Now we can construct a simple graph of two nodes and one edge:

$$\begin{array}{l}\textbf{let } n_1 = \mathsf{node}\ 1\ [\,]\ \textbf{in}\\ \textbf{let } n_2 = \mathsf{node}\ 2\ [(n1, 9.9)].\end{array}$$

We can define a function adding one more edge into a node:

$$\begin{array}{l}(\mathsf{addedge}\ : \mathsf{node}\ X\ Y \to (\mathsf{node}\ (X,Y)) * Y \to \mathsf{comm}) = \\ \mid \mathsf{node}\ x\ l \to (\ \mid e \to \mathsf{primassign}\ l\ (\mathsf{cons}\ e\ (\mathsf{clone}\ l)))\end{array}$$

And now we can add an edge from n_2 to itself:

$$\mathsf{addedge}\ n_2\ (n_2, 2.2).$$

5.5 Implementation

All implementation issues are similar to ones in the previous chapter. The main thing I want to claim here is that all the theory in this chapter has been implemented. Again, all examples have been coded and tested; and they produced expected results. Codes are available on request.

5.6 Conclusion

Like the previous chapter, this chapter is also about extending the pattern calculus with imperative features. This chapter explores a system where locations and their values are of the same type. The decision when to get the value or when to pass the reference is determined inside the evaluation, therefore constructed locations are much easier to use.

In the system, we decide that only data can be stored in a location. Functions are not storable and will be treated as values hence assignment on functions results in exception. Another key point in the design of the system is that a larger location is constructed out of the store variables of smaller locations.

Again, generic functions with imperative features can be written in a simple, efficient and elegant way. The key example is still the generic assignment function where inplace update is used whenever it is possible to do so.

Chapter 6

Generic Data Distribution for Parallel Programming

6.1 Introduction

Data parallelism involves performing a similar computation on many data simultaneously. A common paradigm for data parallelism is a single thread of control containing three stages: data distribution; local operation; and collection. Data distribution divides the data structure into pieces. Local operation performs the calculation on the pieces simultaneously. Collection assembles the local results into a global result. The communication and the synchronization between the stages are usually explicit. That makes data parallelism usually easy to use. However, most of the current work in data parallelism concerns sequential (linear) data structures like lists, arrays. Other work on structured data usually contains a flattening step to change the data structure into a sequential one.

Our approach to parallel programming is to manipulate structured data directly to improve locality. The uses of parametric polymorphism allow one parallel program to act on different data structures, and hence, enable more effort to optimize the programs. As the pattern calculus gives a new powerful account of parametric polymorphism; the expansion of the calculus with parallel features is a great tool for generic parallel programs. As data distribution is an important step in parallel programming, the focus of this chapter is to build a generic data distribution function. We do not get into details of parallel programming such as parallel architecture nor cost models.

6.2 Generic Data Distribution

6.2.1 Type of Resulting Pieces

This subsection discusses the type of resulting pieces of our distribution.

In our research, we have explored an interesting approach: all the pieces are data-structures of the same data as that of the original data-structure but perhaps different structure. In other words, when the original data structure has type $F\ X$ then the resulting pieces have type $F_i\ X$. Hence, when a structure polymorphic function such as map or fold is to be applied the original data structure, the same function can be applied to each piece. One of the advantages of this approach is that it has the potential to divide any data structure into smaller pieces. For example, this approach might divide a pair of a list and a tree into smaller pieces; when the first approach cannot do since there is no substructure of type pair. This approach is challenging since there is no static type for the resulting pieces. Moreover, I could not work on it further since I run out of time. However, this approach is an interesting topic for future work.

Distribution of regular structures such as a list, vector or matrix, the dividing is straightforward. One usually divides lists, vectors or matrices into collections of sub-lists, sub-vectors or sub-matrices. In this thesis, we generalize this usual approach to all data structures; all the resulting pieces have the same type as the original data structure. That implies the original data structure must be recursively defined and must have recursive substructures in order to have a nontrivial distribution. Any data structure without substructure with the same types such that a pair of a list and a tree will be left as one piece (trivial distribution).

6.2.2 Number of Resulting Pieces

To divide a data structure over p processors, we have to break the data structure into n pieces. The n pieces are collected into p subsets, each subset is then assigned to one processor as the result of dividing.

When dividing a list over p processors, a block distribution results in p sub-lists. However, in many cases, the distribution is not that simple. A typical example is to divide a perfectly balanced binary tree over three processors. For simplification, we consider everything in the tree as nodes (leaves of the tree can be considered as nodes with no children). Suppose we divide all the nodes to the three processors; we call all connecting nodes in one processor a piece. If the aim is to have perfect load balance (the numbers of nodes in any two processors are different by at most 1), then it is clear there must be more than 3 pieces. In fact, we will prove that the number of pieces has to be larger than a linear proportion of the depth of the tree as Proposition 11 bellow.

The proposition and its proof below are standard and purely mathematical. We need some definitions and a lemma to make the proof easier. For each natural number n, let $f(n)$ be the number of ones (1) immediately succeeded by a zero (0) when representing n in base two. For example, as 21, 7 and 2 (in base ten) are 10101, 111 and 10 (in base two) we have $f(21) = 2$, $f(7) = 0$ and $f(2) = 1$. For a collection C of pieces of a tree, let $\#C$ be the number of elements of C and let $s(C)$ be the total number of nodes in C.

Lemma 10 *Let C_1, C_2 and C_3 be three collections of pieces as a result of dividing a perfectly balanced binary tree over three processors. We have $\#C_1 + \#C_2 + \#C_3 > f(s(C_1))/4$.*

Proof: The proof of the lemma is by induction on the sum of $\#C_1 + \#C_2 + \#C_3$. Let l be the depth of the tree. The sum of $\#C_1 + \#C_2 + \#C_3$ cannot be less than 1 as the tree must have a root. When the sum is 1 then C_1 is either the empty set or the set of one element which is the whole tree; $s(C_1)$ is 0 or $2^{l+1} - 1$. Then we know $f(s(C_1)) = 0$. Since $1 > 0 = f(s(C_1))/4$; this establishes the base case for the induction proof.

For inductive step, we are trying to have a new distribution D_1, D_2 and D_3 such that

$$\#C_1 + \#C_2 + \#C_3 = \#D_1 + \#D_2 + \#D_3 + 1$$

so we can apply the induction hypothesis on D_1, D_2 and D_3. There is a piece (called S) in C_1, C_2 or C_3 which is a full subtree. So we can create D_1, D_2 and D_3 from C_1, C_2 and C_3 by moving S to join the piece whose nodes include the parent of the root of S. Now apply the induction hypothesis on the three new collections D_1, D_2 and D_3 we have $\#D_1 + \#D_2 + \#D_3 > f(s(D_1))/4$.. When comparing D_1 and C_1 there are three cases: D_1 is the same as C_1; D_1 is the same as C_1 except that one of its pieces gets the extra bit S; or D_1 is C_1 with the piece S removed. Let $t-1$ be the depth of S, the number of nodes in S is $2^t - 1$. We can deduce that $s(C_1)$ is the same as $s(D_1)$ or their difference is $2^t - 1$. Then we know that $f(s(C_1))$ is at most 4 greater than $f(s(D_1))$; combining that with the induction hypothesis would complete the proof of the lemma.

□

Proposition 11 *The number of resulting pieces in distributing a perfectly balanced binary tree of depth l over three processors is greater than $l/8 - 1$ if the distribution has perfect load balance.*

Proof:

If the depth of the tree is odd and equal $2k+1$ then total number of nodes in the tree is $2^{2k+2} - 1$. Having the perfect load balance, the number of nodes for each processors is:

$$(2^{2k+2} - 1)/3 = 2^{2k} + 2^{2k-2} + \cdots + 2^0 = m.$$

We know that $f(m) = k$ and combine that with the lemma, we have the total number of pieces is greater than $k/4$ which is greater than $(2k+1)/8-1$. So, the total number of pieces is greater than $(2k+1)/8-1$.

If the depth of the tree is even and equal $2k$ then total number of nodes in the tree is $2^{2k+1}-1$. Having the perfect load balance, the number of nodes for each processors is m or $m+1$ where:

$$m = (2^{2k+1}-2)/3 = 2^{2k-1} + 2^{2k-3} + \cdots + 2^1.$$

We know that $f(m) = k$ and $f(m+1) = k-1$. The proof is then similar to the case when the depth of the tree is odd.

$\square$

6.2.3 Parameters of Generic Distribution

The distribution function divides any data structure into smaller pieces according to the hardware parameters of the machine. The first argument of the function is the hardware information of the parallel machine e.g. the number of processors, network speed, cache size, etc. The second one is the data structure to be distributed. As the first step in the work, the only concerned hardware parameter is the number of processors.

As the main theme of this work is about data parallelism and data distribution, the focus is on scalable problems [60, 64]. That means we are interested in the problem where the size of the input data is significantly larger than the number of the processors. For example, when the number of processors is larger than the size of input data, the theory does not utilize the fact that we can put each datum into a distinct processor.

6.2.4 Requirements of the Distribution

Since the context of distribution is for parallel programs, the requirements for the distribution are closely determined by the requirements of parallel programs, which are: granularity; load balance; locality; and communication and synchronization [43]. Among these four, load balance and locality largely depend on the choice of distribution and the other two do not. That imposes the two main requirements of distribution: load balance; and locality. By load balance, we mean that all processors hold approximately the same amount of data. Locality means related data tends to be in the same processor, and therefore, potentially reduces communication between processors. For example, locality in distributing a tree is about trying to keep a node and its sub-nodes in the same processor.

However, the requirements sometimes conflict. Distributing a perfectly balanced binary tree helps illustrate the point. The tree can be easily distributed over two, four, or eight processors with perfect locality and perfect load balance. However, when dividing all the nodes of the tree for three processors, we cannot achieve both perfect work balance and perfect locality. As discussed previously, if we want the best possible load balance then the number of resulting pieces has to be proportional to the depth of the tree, i.e. not perfect locality. On the other hand, if we want perfect locality, (the resulting parts are three sub-trees) then the biggest sub-tree is at least double the smallest one, i.e. not perfect load balance.

6.3 Algorithm for Generic Distribution

This section explores an algorithm for the generic data distribution.

As we want all the resulting pieces to have the same type as the original data structure, to have a nontrivial distribution, the original data structure must have recursive substructures with the same type as the original. Distributing a data structure without substructure with the same types leaves the data structure as one piece (trivial distribution). Focusing on non-trivial distribution, in this section, the word *tree* is used to call a recursive abstract datatype with substructures with the same type as the original one. Any immediate substructure with the same type with a tree is called a *subtree* of that tree. [1]

The aim for tree distribution is to break a tree into a set of almost equal parts, each part is a set of sub-trees. Below is an algorithm we have used for tree distribution.

This algorithm traverses through the tree top-down. We need to keep a list (of integers) for the estimated capacity for the amount of data each processor should hold. Initially, the list of estimated capacity has all the same integers, and the sum of all integers in the list is equal (or slightly larger than) the size of data structure that is distributed. The n^{th} element of the list is the amount of data that the processor number n^{th} should take. Each time some data is assigned to the n^{th} processor, the n^{th} element of the list is reduced by the size of the data. In the algorithm below, "small enough" and "best fit" are relations between the size of the data with the

[1]In this section, the term sub-tree represents some sub-structure of the original tree, and subtree (of a sub-tree) represents the immediate sub-structure (of the sub-tree)

current list of estimated work to the processors.

a Small Subtree : If the sub-tree is small enough, (in relation to the work assigned for processors), then assign the sub-tree to the best fit processor. Reduce the estimated work for this processor by the size of the sub-tree.

b Big Subtree : If the sub-tree is not small enough then:

- **b1 Updating Parameters** Put the node (non-recursive part of the sub-tree) on processor zero. Reduce the estimated work for processor zero by the size of the node.
- **b2 Recursive Call** Apply this algorithm on all sub-recursive structures (all subtrees of the sub-tree) using the same list of estimated capacity.
- **b3 Combination of Recursive Call** Combine the results from all the recursive calls.

The actual implementations of "size", "small enough", and "best fit processor" might vary. The size function has the type $X \rightarrow$ int. It counts the number of real data int, float, char. The sub-tree is small enough when there is one processor whose estimated work left is greater than or equal to the size of the sub-tree. The best fit processor is the processor with the least estimated capacity that is greater than or equal to the size of the sub-tree.

In a purely functional approach, we need some mechanism to represent a data structure that belongs in remote (different) processors. One solution is to introduce a new constructor :

$$\mathsf{remote} : \mathsf{int} \to X \to X.$$

For each data x and an integer n, remote n x can be understood as the data x associated with processor number n. In fact, one can treat remote as an advance version of the imperative feature clone. As clone x creates a new copy of x; the ultimate goal of remote n x is to make a new copy of x in the n^{th} processor.

Using the above algorithm, we can traverse through the input data structure, and mark any substructure to be sent to processor n by remote n. Another way to describe that is: distribution is done by marking an appropriate substructure with remote. A version of the main body of the code can be seen in Figure 6.1. From the algorithm described in the previous section, we can see the need for supporting functions for the distribution function. Among supporting functions for the distribution, setChildren and getChildren are two built-in generic functions with the types:

$$\mathsf{getChildren} : X \to \mathsf{list}\ X$$
$$\mathsf{setChildren} : \mathsf{list}\ X \to X \to X.$$

The evaluation of getChildren x will return a list of substructures (children) of x that all have the same type as x. Considering x as a tree then getChildren x returns the list of immediate subtrees of x. The result of the evaluation of setChildren l x is a tree with the root being the same as one x and all children being elements of l. The implementation of these two functions exploits the

```
(dist : list int $\rightarrow X \rightarrow$ list int $* X$) $l$ $x$ =
if smallEnough $l$ (size $x$) then
  let $n$ = mostFit (size $x$) $l$ in
  (updateEntry $n$ (fun $s \rightarrow s -$ (size $x$)) $l$, remote $n$ $x$)
else
  let $l_1$ = updateEntry 0 (fun $s \rightarrow s -$ (sizeTop $x$)) $l$ in
  match (paramaplist dist $l_1$(getChildren $x$)) with
   | $(l_2, x_c) \rightarrow (l_2$, setChildren $x_c$ $x$)

(distribution : int $\rightarrow X \rightarrow X$) $p$ $x$ =
snd (dist (sameList $p$ ((size $x$)/$p$ + 1)) $x$)
```

Figure 6.1: Generic Tree Distribution

theory of representation of data; and we are not going to details of those in this thesis. However, the compiler code for the two functions as well as one for remote are available on request. We also need some standard functions acting on lists: mostFit; updateEntry; sizeTop; paramaplist; sameList and smallEnough. The exact FISh2 code for all auxiliary functions is given bellow.

```
let rec (size: X -> int) =
| int x -> 1 | float x -> 1 | char x -> 1
| x y -> (size x) + (size y)
| x -> 0;;

let rec (smallEnough: list int -> int -> bool) l x =
match l with
| nil -> false
| cons h t -> if h >= x then true else smallEnough t x;;

let (mostFit: int -> list int -> int) x l =
  let rec (gmf: (int*(int*int)) -> list int -> (int*int)) i =
  | nil -> (snd inpr)
  | cons h t -> (
  if ((h < x) || ((snd (snd i)) <= h)) then
    (gmf ((fst i) + 1, snd i) t)
  else
    (gmf ((fst i) + 1, (fst i, h)) t))
in fst (snd (gmf (0, (-1,x)) l));;

let rec (updateEntry: int -> list X -> (X -> X) -> list X)
n l f =
match l with
| nil -> nil
| cons h t -> if (n = 0) then (cons (f h) t)
    else (cons h (updateEntry (n - 1) t f));;

let appsnd f pr = (fst pr, f (snd pr));;

let rec (paramaplist: (X->Y->X*Z) -> X -> list Y -> X*(list Z))
f i =
| nil -> (i, nil)
| cons hd tl -> let t = f i hd in
appsnd (cons (snd t)) (paramaplist f (fst t) tl);;

let rec (sameList: int -> X -> list X) n x =
if n < 1 then nil else Cons x (sameList (n - 1) x);;

let rec (sumSize: list X -> int) =
| nil -> 0
| cons h t -> (size h) + sumSize t;;

let (sizeTop: x -> int) = size x - (sumSize (getChildren x));;
```

6.4 Implementation

The generic data distribution was developed in 2002 as part of a parallel mapping function as a join work of Jay [2], Hamdan [3] and me. At that time, FISh2 supported some parallel MPI primitives [62, 61] but they have not

[2]C. Barry Jay University of Technology, Sydney, Australia
[3]Mohamamd M. Hamdan, Yarmouk University, Irbid, Jordan

been maintained since 2002. We run some experiments on pmap but did not get any significant results. I implemented the generic data distribution function as discuss in the previous section. The function had been tested (it is a sequential algorithm and can be run on one processor) and it produced expected results on different kinds of data such as lists and various sorts of trees. The code has been maintained and presented above. Most of work is about writing FISh2 codes. I also added all built-in constructors and functions needed for the distribution such as remote and getChildren into the compiler. The compiler with those features is available on request.

6.5 Conclusion

Generic programming promises an effective tool for parallel programming. Generic data distribution is a key step in building generic parallel programs. We have found a new generic algorithm for distributing structured data with all the resulting pieces having the same type as the original data structure. The algorithm tries to maximize the load balance while preserving major part of the structures.

Chapter 7

Conclusions

The pattern calculus, as introduced by Jay [35] 2004, provides a powerful and new approach to generic functions. The expressive power of the calculus mostly depends on the choice of the constants. We increase its expressive power further by extending (or/and changing the properties of) the constants to cover different styles of programming.

In designing the type system to support both polymorphism and imperative programming, we need to ensure that the two features do not interact in any ill-typed way. We use value polymorphism approach, which contains simple type rules to get type safety in systems containing both polymorphism and updatable location.

Parametric polymorphism can be safely integrated into an imperative system in a simple way. The key point is to separate functions which are potentially polymorphic from updatable terms. We use the polymorphic let rule for extensions (functions) or variables only, and use the monomorphic let rule for other terms (and the same for fix-point). The two let rules are the way for combining polymorphic functions and monomorphic updatable data into one small simple system. As a result, all functions in the system can be used fully polymorphically in conjunction with updatable locations.

We have two different expansions of the calculus with imperative features. They combine the functional and imperative programming styles within a single, simple calculus.

The first expansion adds more expressive power to the system with a minimum effect on the purely functional features. There is a clear separation between the type of a location loc T and the type of its value T. Based on the powerful technique for building generic functions of the pattern calculus, primitive imperative operations can be used to underpin generic operations for creating, reading from and writing to constructed locations. The advantage of this approach is that it naturally supports inplace update, something that is hard to achieve in other languages supporting recursive types. By adding a layer of indirection, one can wrap inplace update within a completely safe form of assignment on reference types. The added features of while-loops and for-loops, vector types and the generic output function is one proof for the orthogonality of the systems. The addition of loops and vector types shows the potential for optimizing the system without affecting its soundness. The addition of generic output function also shows that exploiting the expressive power of the system can help to reduce the complexity of the compilers.

The second expansion adds the update ability to storable terms. A key feature of the system is that there is no boundary between a location and its value. The decision when to get the value or when to pass the reference are all determined inside the calculus, therefore constructed locations are much easier to use. As in the previous expansion, we can construct a generic assignment function which naturally supports inplace update.

Generic programming promises an effective tool for parallel programming. The expressive power of the pattern calculus can also be exploited in parallel programming. As the generic assignment is an effective tool for handling locations; the generic data distribution plays similar roles in parallel programming. Generic data distribution is a key step in building generic parallel programs. We have found a new generic algorithm for distributing structured data. The algorithm tries to maximize the load balance and the locality of data.

7.1 Future Work

The focus of this thesis has been to extend the expressive power of the pattern calculus in an imperative setting. It can be used as a base to add extra features and find more applications. The main future directions are summarized as follows.

Finding More Applications The combination of the pattern calculus and imperative features promises great expressive power. We can try to find some real world applications where the expressive power is utilized and magnified. The process might help in identify more useful built-in functions.

Adding the Parallel Features An obvious direction for future work is the adding of parallel features to the extension of pattern calculus with imperative feature. One possible further development is of generic skeleton (see e.g. [18]). One of the steps, a generic distribution function has been outlined in this thesis. The communication between different processors can be seen as assignment of data from one processor to another.

Object-Orientation Another direction is finding a new approach to object-orientation. With the ability to define functions with different algorithms for different types, the pattern calculus is proved to be a new promising base for object-oriented languages [34]. Adding that with the result of this thesis should create a uniform system that supports functional, imperative and object-oriented programming styles.

Database Programming Adding imperative features is an important step in applying the pattern calculus to database programming [48]. In some way, databases can be seen as large constructed locations. With the new expressive power from the pattern calculus, especially the generic assignment function as an efficient way to deal with large locations, we might have a new way to act on databases in a generic way.

Bibliography

[1] Andrew W. Appel. A critique of standard ML. *Journal of Functional Programming*, 3(4):391–429, 1993.

[2] J. E. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324(6270):446–449, 12 1986.

[3] R. Bird and P. Wadler. *Introduction to Functional Programming.* International Series in Computer Science. Prentice Hall, 1988.

[4] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press and Pitman, 1989.

[5] Murray Cole. List homomorphic parallel algorithms for bracket matching. Technical Report 93/29, University of Edinburgh, 1993.

[6] Karl Crary. Sound and complete elimination of singleton kinds. *Lecture Notes in Computer Science*, 2071:1–26, 2001.

[7] Jacques Chassin de Kergommeaux, Philip J. Hatcher, and Lawrence Rauchwerger. Parallel computing for irregular applications. *Journal of Parallel Computing*, 26:1681–1684, 2000.

[8] A. Dimock, I. Westmacott, R. Muller, F. Turbak, J. Wells, and J. Considine. Space issues in compiling with intersection and union types. In *ACM SIGPLAN Workshop on Types in Compilation*, 2000.

[9] Bruce F. Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ML. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 163–173, Orlando, Florida, 1991.

[10] Dominic Duggan. Sharing in typed module assembly language. *Lecture Notes in Computer Science*, 2071:85–116, 2001.

[11] A. J. Field and P. G. Harrison. *Functional Programming.* Addison-Wesley, 1988.

[12] Ian Foster. *Designing and Building Parallel Programs.* Addison-Wesley Publishing Company, 1995.

[13] Daniel P. Friedman, Christopher T. Haynes, and Mitchell Wand. *Essentials of programming languages.* Massachusetts Institute of Technology, 1992.

[14] David Stoutamire Gilad Bracha, Martin Odersky and Philip Wadler. GJ specification. ongoing work, 1998.

[15] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[16] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Third Edition.* Addison-Wesley, Boston, Mass., 2005.

[17] E. Gutirrez, Rafael Asenjo, O. G. Plata, and Emilio L. Zapata. Automatic parallelization of irregular applications. *Journal of Parallel Computing*, 26:1709–1738, 2000.

[18] Mohammad. Hamdan. *A Combinational Framework for Parallel Programming Using Skeletons.* PhD thesis, Department of Computing and Electrical Engeneering, Heriot-Watt University, Edinburgh, January 2000.

[19] K. Hammond and G. Michaelson, editors. *Research Directions in Parallel Functional Programming*, chapter Shaping Distributions. Springer, 1999.

[20] Michael Hicks, Stephanie Weirich, and Karl Crary. Safe and flexible dynamic linking of native code. *Lecture Notes in Computer Science*, 2071:147–176, 2001.

[21] J. Roger Hindley. *Basic Simple Type Theory.* Cambridge University Press, 1997.

[22] Ralf Hinze. A new approach to generic functional programming. *Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 27, 2000.

[23] Ralf Hinze, Johan Jeuring, and Andres Luh. Type-indexed data types. *Proceedings of the Sixth International Conference on Mathematics of Program Construction*, 6:148–174, 2002.

[24] Gerard Huet. The zipper. *Journal of Functional Programming*, 7:549–554, 1997.

[25] R.J.M. Hughes, L. Pareto, and A. Aiken. Proving the correctness of reactive systems using sized types. In *Symposium on Principles of Programming Languages*. ACM Press, 1996.

[26] Patrik Jansson. *Functional Polytypic Programming.* PhD thesis, Computing Science, Chalmers University of Technology and Gteborg University, Sweden, May 2000.

[27] C. Barry Jay. bondi web-site. `www-staff.socs.uts.edu.au/~cbj/bondi/`.

[28] C. Barry Jay. FISh web-site. `www-staff.socs.uts.edu.au/~cbj/FISh/`.

[29] C. Barry Jay. Shape analysis for parallel computing. In J. Darlington, editor, *Proceedings of the fourth international parallel computing workshop: Imperial College London, 25–26 September, 1995*, pages 287–298. Imperial College/Fujitsu Parallel Computing Research Centre, 1995.

[30] C. Barry Jay. Separating shape from data. In E. Moggi and G. Rosolini, editors, *Category theory and computer science: 7th international conference, CTCS'97, Santa Margherita Ligure, Italy, September 1997 Pro-*

ceedings, volume 1290 of *Lecture Notes in Computer Science*, pages 47–48. Springer Verlag, 1997.

[31] C. Barry Jay. The FISh language definition. `www-staff.socs.uts.edu.au/~cbj/Publications/fishdef.ps.gz`, 1998.

[32] C. Barry Jay. Costing parallel programs as a function of shapes. *Science of Computer Programming*, pages 207–224, 2000.

[33] C. Barry Jay. Distinguishing data structures and functions: the constructor calculus and functorial types. In S. Abramsky, editor, *Typed Lambda Calculi and Applications: 5th International Conference TLCA 2001, Krakó w, Poland, May 2001 Proceedings*, volume 2044 of *Lecture Notes in Computer Science*, pages 217–239. Springer, 2001.

[34] C. Barry Jay. Methods as pattern-matching functions. In *Foundations of Object-Oriented Languages, 2004: informal proceedings*, page 16 pp, 2004. `http://www.doc.ic.ac.uk/~scd/FOOL11/patterns.pdf`.

[35] C. Barry Jay. The pattern calculus. *ACM Transactions on Programming Languages and Systems*, 26(6):911–937, 2004.

[36] C. Barry Jay, Gianna Belle, and Eugenio Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, 1998.

[37] C. Barry Jay H.Y. Lu, and Q.T. Nguyen. The polymorphic imperative: a generic approach to in-place update. In Mike Atkinson, editor, *Computing:The Australasian Theory Symposium 2004*, volume 91

of *Electronic Notes in Computer Science*, pages 192–206. Science Direct, 2004.

[38] S.L. Peyton Jones. *Haskell 98 Language and Libraries.* Cambridge University Press, Cambridge, UK, 2003.

[39] Gabriele Keller and Manuel M. T. Chakravarty. Flattening trees. In *Proceedings of the 4th International Euro-Par Conference (EuroPar98)*, Lecture Notes in Computer Science, pages 709–719. Springer Verlag, 1998.

[40] Gabriele Keller and M. Simons. A calculational approach to flattening nested data parallelism in functional languages. In J. Jaffar and R. H. C. Yap, editors, *Concurrency and Parallelism, Programming, Networking, and Security: Second Asian Computing Science Conference, ASIAN'96*, volume 1179 of *Lecture Notes in Computer Science*, pages 234–243. Springer Verlag, 1996.

[41] Xavier Leroy and Pierre Weis. Polymorphic type inference and assignment. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida*, pages 291–302. ACM Press, 1991.

[42] M. Leucker, T. Noll, P. Stevens, and M. Weber. Functional programming languages for verification tools: Experiences with ml and haskell. In *Proceedings of the Scottish Functional Programming Workshop (SFPW'01)*, 2001.

[43] Ted G. Lewis and Hesham Rl Rewini. *An Introduction to Parallel Computing.* Prentice-Hall, 1992.

[44] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17, 1978.

[45] Robin Milner and Mads Tofte. *Commentary on standard ML.* MIT Press, 1991.

[46] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[47] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[48] Clara Murdaca and C. Barry Jay. A relational account of object. ongoing work, 2005.

[49] Nesl home-page. URL: `www-2.cs.cmu.edu/~scandal/nesl.html`.

[50] Susumu Nishimura and Atsushi Ohori. Parallel functional programming via data-parallel recursion. *Journal of Functional Programming*, 9(4):427–463, 1999.

[51] Objective Caml home page. `pauillac.inria.fr/ocaml`.

[52] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 146–159. ACM Press, New York (NY), USA, 1997.

[53] Alvaro J. Rebon Portillo, Kevin Hammond, Hans-Wolfgang Loidl, and Pedro B. Vasconcelos. Cost analysis using automatic size and time inference. In *Lecture Notes in Computer Science 2670*, pages 232–247. Springer, 2003.

[54] Sven-Bodo Scholz. Single Assignment C - functional programming using imperative style. In *Functional Languages Implementation Workshop, Norwich, UK, 1994*, 1994.

[55] Robert W. Sebesta. *Concepts of programming languages.* Addison-Wesley, 1996.

[56] Ravi Sethi. *Programming languages concepts and constructs.* Addison-Wesley Publishing Company, 1996.

[57] David B. Skillicorn. *Foundations of Parallel Programming.* Number 6 in Cambridge Series in Parallel Computation. Cambridge University Press, 1994.

[58] David B. Skillicorn. Parallel implementation of tree skeletons. *Journal of Parallel and Distributed Computing*, 39:115–125, 1996.

[59] David B. Skillicorn. Questions and answers about BSP. Technical Report 96/25, Oxford University Computing Laboratory, 1996.

[60] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Computing Surveys (CSUR)*, 30(2):123–169, 1998.

[61] Marc Snir, Steve W. Otto, David W. Walker, Jack Dongarra, and Steven Huss-Lederman. *MPI: The Complete Reference.* MIT Press, Cambridge, MA, USA, 1995.

[62] Jeffrey M. Squyres. Definitions and fundamentals – the message passing interface (MPI). *ClusterWorld Magazine, MPI Mechanic Column*, 1(1):26–29, December 2003.

[63] Standard ML of New Jersey home page. `cm.bell-labs.com/cm/cs/what/smlnj/`.

[64] H. Stockinger, K. Stockinger, E. Schikuta, and I. Willers. Towards a cost model for distributed and replicated data stores. In *9th Euromicro Workshop on Parallel and Distributed Processing*, pages 461–467. IEEE CS Press, 2001.

[65] R.D. Tennent. *Semantics of Programming Languages.* Prentice Hall, 1991.

[66] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1), November 1990.

[67] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52, 1995.

[68] K. Wansbrough and S. Jones. Simple usage polymorphism. In *ACM SIGPLAN Workshop on Types in Compilation*, 2000.

[69] Andrew K. Wright. Polymorphism for imperative languages without imperative types. Technical Report TR93-200, Rice University, 1993.

[70] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.

[71] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[72] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of Programming Language Design and Implementation(PLDI '98), Montreal, June 1998.*, pages 214–227, 1998.