Abstraction-level Functional Programming

Allan Clark

Doctor of Philosophy
Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh
2008

Abstract

This thesis is concerned with abstraction-level programming, where abstraction-level is the level of programming tasks which extend the abstraction of the machine. Extending the abstraction of the machine is generally done by compiler writers for high-level programming languages or those implementing an interface to lower-level or legacy libraries. The abstractions which are implemented are then used, either explicitly or implicitly, by the high-level language programmer. The main aim of the abstraction is often to increase programmer productivity but can also be for efficiency or security reasons. Implementing an automatic runtime garbage collector is a common example of an abstraction-level programming task.

To date most abstraction-level programming has been done in low-level programming languages such as C. The contents of this thesis describes an investigation into the design of a functional language Nitro, for use in abstraction-level programming. The main goal is to provide the abstraction-level programmer with some of the benefits enjoyed by high-level functional language programmers.

Acknowledgements

First and foremost my thanks to Stephen Gilmore who has provided optimum supervision and technical input and doing so in the most friendly of manners. David Aspinall has provided well received technical input as well as being continually available. John Longley has provided advice and support and in addition as always made himself available at the shortest of notice. It is thanks to John Longley that I first looked at the problem of inferring exception annotations for functional programs and eventually applied this to the delayed typing scheme described in this thesis. My thanks as well to my parents who have never wavered in their support of any of my decisions both academic and personal (and in other areas where my decision making abilities are much more questionable). Friends too numerous to mention have provided many more relaxed moments and allowed me to retain a good perspective. Tim Jones and Andy Koppe have been interesting and accommodating office-mates both providing technical assistance whenever it was required.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Allan Clark*)

Table of Contents

Chapter 1

Introduction

The main purpose of a programming language is to allow humans to communicate instructions to a computer such that it can perform some repetitive time-consuming task with high reliability and speed. Languages have been developed to reduce the risk that the programmers make mistakes when instructing the computer. Languages are now closer to natural language than previous generations of programming languages were, while at the same time maintaining the property of non-ambiguity.

This allows the increased documentation and readability of programs which is a constant area of research aimed at reducing the errors made by programmers. However, inevitably some programs will contain errors. Modern programming languages contain facilities for the re-use of code such that portions of programs that are correct can be re-used by other programs. When a program contains an incorrect portion if such a portion is re-used then it is more likely that the deficiency will be found. In addition when such a defect is corrected it is corrected for all programs which have re-used the common functionality.

1.1 Strong typing vs Weak typing

A further area of research is the automatic detection of programmer errors. Types are now commonly used for this purpose. Entities within a program are given types by the type system which forbids the use of an entity inconsistent with the type it has been given. For example the multiplying together of two objects makes sense only when the types of the two objects are numerical, or have, in some way, been associated with a multiplication operator. In this way type systems aid the compiler in rejecting code that is unlikely to do what the programmer has intended it to. Furthermore it can reject some code that may be correct but is difficult to maintain and/or reuse. When a strong type system is used it can be coupled with an information hiding mechanism. Implementation details can be hidden so that relatively distant parts of the code cannot be programmed to rely on a specific implementation detail. This increases maintainability and re-usability because changing one part of the code will, in many cases, not require a change to another and in the cases that it does the compiler can reject the program until all dependent parts are modified consistently.

The term *strong typing* has several meanings including, whether or not variables within a program can change their type, or the degree to which implicit type conversions are allowed. In this thesis the most general notion of strong typing is applied, this concerns whether or not such type errors as those described in the preceding paragraph are allowed within the user program. A language using strong typing does not allow the programmer to use a value of one type as though it were of another incompatible type. With *weak typing* this does not hold and erroneous programs are accepted by the compiler. In weakly-typed languges in some cases warnings are issued by the compiler and external program analysis tools are used to help locate bugs that the weak type system could not find, for example[1, 2, 3].

1.2 Static vs Dynamic typing

The issue of strong and weak typing is not the same as *static* and *dynamic* typing. These terms refer only to when the typing occurs. A static type system type checks the program before it is run while a dynamic type system checks the types of values at runtime immediately before a type dependent operation is executed. One can have any combination of the two distinctions of type system: Static-Weak, Static-Strong, Dynamic-Weak and Dynamic-Strong.

A static-strong type system aims to reject all programs that are guaranteed to result in a runtime type error (for at least some inputs) and allow all other programs. Because this is not in general possible a static strong type system is usually a conservative approximation; it will not allow a type incorrect program to be run but might also reject some programs which would never fail with a type error.

For some of the type correct programs rejected by the compilers for popular static-strongly typed programming languages it is desirable that the type system rejects them because they are not maintainable, however there are some programs rejected which would be useful to execute[4].

A dynamic type system will allow both more correct and incorrect programs to be run. This means that some correct programs that cannot be type checked statically are allowed to run but also that finding faults in programs must be done by running tests. Some programming languages incorporate a compromise between the two such as the notion of *soft typing*[5].

A certain class of errors cannot be detected at compile-time and must be deferred to run-time. In this case the earlier that the error is detected the less damage can result from the error and the easier it is for the programmer to correct their program. Runtime error detection usually involves a runtime cost in the efficiency of the program. This means that although a type system is classified as a static type system there are some dynamic type checks which must be performed. A very common dynamic check included within static type systems is the check inserted prior to an array index operation which ensures that the index is within the bounds of the array.

1.3 Functional vs Imperative

Languages described as *imperative* are so called because a program written in such a language consists of a sequence of commands which are executed in order. An imperative language may include other features to assist in composing the sequence of commands, for example an object oriented language may also be an imperative language. Examples of imperative languages include C and Java [6].

The term *functional language* can be given several different meanings. Commonly there are two features which are associated with the meaning, these are: higher-order functions, and a lack of side-effects such as destructive update or assignment.

To enable higher-order functions, functions must be first class values within the language. This means that a function must be considered as a normal value and in particular can be passed around as an argument or returned as the result of another function. Imperative languages often include this feature. Although C is generally regarded not to contain first class functions the programmer may pass in or return a function pointer, though these may not be created at runtime. Higher-order functions combine functions as first class values together with nested functions. A nested function is a function whose definition is contained within the definition of another function. Purely syntactically nested functions though are not really nested. For a function to be truly nested it must use a variable or value defined outside of its own definition and within the definition of the function within which it is nested. If this is allowed then a value can escape its scope, because the nested function which references it can be passed out as part of the result of the function in which it is defined.

In this thesis the term functional language will be taken to mean a language which provides higher-order functions and discourages side-effects. A pure functional language will be taken to mean a language which entirely forbids expressions with side-effects.

1.4 High-level vs Low-level languages

Programming languages can be distinguished by the level of abstraction from the details of the machine that they provide. In high-level programming languages the details of the machine are kept hidden from the programmer. The programmer describes how to solve the problem in a way which is independent of the machine on top of which the compiled program will run. In low-level programming languages such details are available to the programmer.

A high-level language aims to abstract into programming language constructs a set of useful operations and/or idioms. The main goals are to increase the readability of the program, increase the productivity of the programmer and increase the amount of information about the program that the compiler can infer in order that it may detect more errors in the code. These three aims give clues as to what kind of operations or idioms should be abstracted into a language construct. Anything which is repeated often enough and/or can be a source of program errors may be represented better as a language construct. By using a higher-level construct the programmer is conveying more of the intention of the code to the compiler. For example an enumeration type conveys to the compiler that it is only appropriate to store a small set of values in any location given that type. Because the compiler knows this it can detect any attempt to assign a value outside that range of small values.

Furthermore a high-level language which imposes a strong type system must provide a construct for some operations that could otherwise not be expressed within the type system of the language. An example of this is an exception mechanism – an arbitrary jump out of a function is generally not allowed within a high-level programming language and hence exceptions are commonly provided as a language construct.

Some common abstractions include:

- Tagged union data types
- Objects
- Checked array representations

- Record and tuple values
- An exception mechanism
- Garbage collection
- Strong typing

In contrast a low-level language does not provide abstract constructs which hide implementation details. Often the constructs provided are for the convenience of the programmer only. The programmer is granted access to the details of the machine. Such access allows optimisations based on the representation of data to be included in the program. A low-level language often provides no automatic runtime garbage collection and offers a weak type system. Because the type system is weak and the programmer has control over the representation of their data it is possible for the programmer to reuse a block of memory for a new value of a different kind from the one for which the memory block was previously used. With this ability comes the responsibility to make sure that the only memory blocks reused in this fashion are those which will never again be accessed as the old value. In other words it is the programmer that must make sure never to reuse memory in which a live value is stored.

1.4.1 Advantages of the higher abstraction

The programmer is not burdened with the details of the machine, and can therefore concentrate fully on the task of solving the given problem. Details such as how values are stored and when they are relinquished are not necessary for the implementation of many programming problems. Productivity is increased as the compiler has more information about the intentions of the programmer and can therefore warn or even reject code that is likely (or guaranteed) not to perform the intended computation. Programmers therefore spend less time fixing faults in their programs. In addition parts of the program, for example the management of the memory, are already implemented within the programming language and hence the task is reduced before the programmers even begin to write any code.

1.4.2 Advantages of the lower abstraction

Programmers have wide-ranging control over the entities within their programs. With the control comes a degree of predictability. For example one can know in advance when a given portion of memory will be deleted, and in the absence of run-time services such as garbage collectors there is a more straightforward relationship between program runtimes predicted from the source code and measured runtimes.

Such predictability is frequently required when the programming task concerns the operation of the machine itself, rather than the use of the machine to perform a universal computation.

1.4.3 High and Low-level tasks

Both high-level and low-level languages have their advantages and disadvantages and the choice of programming language for any given task is dependent on many things. Some programmers like to feel that they are in complete control, even if that control is not necessary for their given problem. However I would like to submit that programs can be classified into two separate groups which characterise well the use of high-level and low-level languages.

The key distinction is that either the programmer is extending the interface of (or abstraction to) the computer or they are using the interface to the computer. For example a garbage collector: either the programmer is implementing the garbage collector and thus extending the abstraction to the computer, or is using the garbage collector to take care of the details of managing the storage of values within their program.

In this thesis a further distinction is made between different kinds of low-level, or abstraction-providing, programming. Where the programmer must directly access the machine (for example when implementing perhaps of the operating system kernel) this is defined to be low-level; where this is not necessary this is defined to be abstraction-level. Code which is low-level is generally machine dependent whereas code which is abstraction-level is either machine independent or easily portable. It is at this level where it is hoped that more of the beneficial features provided to the high-level language programmer can be transferred to the lower-level programmer, in this case the abstraction-level programmer.

1.5 The Abstraction Type Gap

Ignoring the issue of efficiency for the present, it is clear that strong typing is a useful abstraction. Providing the type abstraction however, commonly involves implementing part of it outside of the type abstraction itself. Consider the case of checked array accesses. The type system should make sure that whenever an array is accessed the index used is greater than or equal to zero but less than the length of the array. If this is not the case then the value returned will not be part of the array and therefore cannot be guaranteed to be of the correct type. A simple way to achieve this kind of type safety abstraction is to use a dynamic check. The length of the array is stored together with the array, then whenever an index operation is invoked the index is checked against the length stored together with the array. There must be some way to check that the length stored with the array is in fact the real length of the array, and that when an access is made the required check is done. This can be done by implementing the creation, indexing and updating of arrays outside of the target language, and providing no access to the programmer to the bare array format. The only access that the user has to the abstract arrays is through the three exposed operations. Therefore the user cannot make a mistake by misusing the representation of the array since it is not known to them. The implementation of these three operations themselves though must be written without this abstraction.

For such low-level untyped code to be trusted, aside from trusting the programmer to have made no mistakes, the compiler of the higher-level language is trusted to retain invariant properties of values which the generated code creates. In the example of the arrays the compiler is trusted to produce code which only invokes the operation for indexing an array with a value which was created with a call to the create array operation. More generally every argument to a call of the index array operation will be in the format expected with the integer which should represent the length of the array actually being equal to that length.

The assumption that the abstraction-level programmer has made no mistakes is a trust that is often acquired through means of testing. This is because such abstraction-level programming is often done in the low-level weakly-typed language C [7]. Suppose that the three array operations defined above are implemented in C, this allows the programmer to access an array with any index whatsoever. There is nothing to ensure that the index is checked against the length of the array.

These are two distinct trusts that are made when using abstraction code written in a separate language: firstly that the assumptions made by the abstraction code about the manner in which the abstractions are invoked are upheld by the compiler for the high-level language; secondly that those assumptions are used correctly by the abstraction code itself. The separate elements that must be trusted are expanded upon in the next section.

1.6 Trust

If any given application code is to be trusted to be correct, or at least safe to run, then there are several parts that must be separately trusted.

1.6.1 The application code

The application code itself must be correct. Confidence in the safety and correctness of the application code can be increased if it is written in a high-level strongly-typed language. The type system will ensure that the application code contains no type errors so the program code is safe to run (though it may still be incorrect in the sense of returning the wrong values). In this case the application code need not be trusted to be safe as it is checked automatically, therefore by using a type-safe language the amount of code that must be trusted has been reduced.

1.6.2 The Compiler

Using a high-level language allows some guarantees about the application code to be checked mechanically by the compiler, but some of those guarantees only hold if the compiler is itself guaranteed to be correct. It is not enough to trust the compiler to be safe, the compiler must be trusted to be correct.

The type system makes sure that the application code does not make any invalid memory accesses if the type system has been proven to enforce such a property, but only if the compiler correctly implements the given type system. Even if the compiler's type checker has been correctly implemented, perhaps the translation into machine code is faulty. This could mean that correct or safe application code is transformed into incorrect or unsafe machine code.

1.6.3 The Abstraction Code

The application code written in the high-level language and translated into machine code uses – implicitly or explicitly – the abstraction code that extends the interface of the machine. Therefore the abstraction code itself must also be correct. The abstraction code must generally be programmed in a low-level programming language. If the abstraction code is programmed in a strongly-typed language then, as is the case with strongly-typed application code, no trust need be placed on the safety of the abstraction code. Once again the trust must instead be placed on the compiler for the language used. Because the abstraction code commonly implements operations that are difficult to ensure the safety of, abstraction code is commonly implemented in a weakly-typed programming language.

1.6.4 The Compiler/Abstraction-Code Link

Finally the compiler for the application code produces machine code that utilises the abstraction code. Therefore the compiler and the abstraction code must have the same set of assumptions about the way in which the abstraction code is invoked.

There is a protocol or interface that the abstraction code must provide and the compiled code uses. Often this protocol is maintained by informal methods, occasionally parts of this protocol can be type checked. Many of the invariants assumed by the abstraction must be maintained by the authors of the compiler by hand. If the entire application is to be trusted to be safe, then this link between the compiler and the abstraction code must also be trusted.

The remainder of this thesis describes the development of a type-safe functional language Nitro. Nitro is designed to be used for abstraction-level programming in order that the programmer may have more confidence in the safety and correctness of the abstraction code.

1.7 Introducing Nitro

Nitro is a programming language developed to survey the ideas contained within the current thesis. The main goal is to provide a functional language that can be used to implement the abstraction code which the implementations of other high-level (functional as well as non-functional) languages rely upon. Writing such abstraction code is often seen as low-level programming, however in order to be clear in this thesis such abstraction code programming will be referred to as *abstraction-level* programming. The aim in developing Nitro is to bring many of the advantages of high-level functional programming to the aid of the abstraction-level programmer.

1.8 Abstraction-level Examples

This section introduces the primary motivating examples of abstraction-level programming for which Nitro is intended to be used. These three examples provide features which are very commonly found in high-level programming languages, but which are most often implemented in C.

1.8.1 Generic Primitives

In order to assist in the writing of the abstraction-level code the compiler writers for a given language will ensure that all values within a program conform to some common internal representation. For example the language designers may wish to provide a single polymorphic equality function which operates over two values of any one type. It is possible to write an equality operator if the common internal representation has some simple properties. These are:

- All values are represented by one common sized element for example a word of the machine.
- A value can be tested to determine whether it is itself immediately the value – as is the case for an integer – or if the value represents a pointer to a block of values in memory. This can be determined using an `is_block` operation.
- A block of values in memory is always attached with a special value which represents the length of the block in values. In this case the primitive – `block_length` – can be used to determine the length of the block.
- Blocks may be indexed with integer offsets where each offset is itself a value in the common internal representation.

Using these assumptions and mapping the block indexing operation to an array-like syntax such as: `b[i]`, where `b` is a block pointer and `i` is an integer offset; the generic equality operator can be written as in figure 1.1.

```
equality: value1 value2 =
  if is_block value1
  then if is_block value2
       then equality_block value1 value2
       else false // value1 but not value2 is a block
  else if is_block value2
       then false // value2 but not value1 is a block
       else value1 == value2

equality_block: block1 block2
  if block_length block1 == block_length block2
  then forall (i = 0 to (block_length block1) - 1)
          equality block1[i] block2[i]
  else false // blocks have different lengths
```

Figure 1.1: Generic equality code

Notice that there is a potential gap between what the abstraction-level code considers to be equal and what the language defines as being equal. Consider the equality test **`true`** `= 1`, if the type system allows such an expression then the language definition must consider what the result should be. The language definition might reasonably define the result of comparing values of different type to either be **`false`** or to raise an error – here we suppose that the result should be **`false`**. However if the two values share the same representation then the given generic equality operator will return **`true`**. If this definition is to be used then it is up to the compiler writer to make sure that two values considered to be distinct by the language specification are either given distinct representations or are not allowed to be tested for equality. In high-level languages this is commonly done by the type system insisting that the type of the two operands are the same and hence two values of different types are never compared.

1.8.2 Garbage Collection

Implementing a garbage collector has similar problems to overcome. It must be possible to see all values within a program as belonging to some common type, otherwise memory allocated for one value cannot be re-used to store another value without arbitrarily changing the type of a storage location. Furthermore it must be possible to distinguish between pointer and non-pointer values such that the roots of a program may be followed to find the live data.

There must be a mechanism within the language used to implement the garbage collector to manage the garbage collector's own memory. While computing the memory management of the application code the garbage collector will itself produce some data which must be relinquished appropriately. There has been much research (for example [8, 9]) devoted to achieving this in a type-safe manner and it is proving to be very challenging.

1.8.3 Marshalling

If a high-level language is to see popular usage then it must provide access to libraries written in other languages. This allows the use of legacy code as well as new code that is simply not written in the desired language. However code written in another language will produce values that will not conform to the common internal representation of the high-level language. This means that the use of generic primitives such as the equality operation defined above is not possible. The foreign values, those produced by the other languages, must also be separated out from the native values. This prevents the garbage collector and other runtime services attempting to interpret the foreign values as values conforming to the common internal representation. Such an error could lead to the invalid dereferencing of a non-pointer value.

Rather than separating such foreign values from native values another approach is to simply provide a conversion routine which converts from the foreign format into the native representation. Unfortunately though, because a high-level language abstracts away from the runtime representation of values the programmer cannot access the raw representation of the foreign values. Therefore the programmer must rely on a translation routine written in an abstraction-level programming language which can *'see'* into the foreign internal representation.

This has a runtime performance cost because each value must be structurally translated into the native representation, however the advantage is the ability to perform all operations provided by the native language and no separation of foreign values from native values is necessary. Translated values can be created within the garbage collected portion of memory and thus the significant abstraction of automatic memory management is maintained even when interfacing with a foreign language.

These three examples of abstraction-level programming are very often performed in the low-level language C. These tasks can generally not be performed within the high-level language while maintaining representation abstraction. However the tasks themselves are not low-level from the point-of-view of their implementation. They are only low-level from the point-of-view of their use. A similar distinction is made when implementing a compiler. The compiler will generally produce low-level assembly or even machine code. However the compilation itself is simply a high-level computation from some input – the source program – to some output – the compiled program. The main point is that access to the representation of the high-level program's values is required, but low-level access to the machine is not.

1.9 General Approach

The general approach to provide abstraction-level programmers with high-level language features is to enhance the type system such that more information can be provided by the programmer to the compiler via the type system. Data types will be used as they are used in both low-level weakly-typed languages and high-level strongly-typed languages. That is, the type of a value both describes how it can be represented in memory and prevents the misuse of the value at an incompatible type.

The type system is then further enhanced to describe more precisely the behaviour of code operating on values given such types. This allows code that would not be passed by type systems in common usage for high-level languages to be written in Nitro. This includes providing the programmer with the ability to describe within the type of an argument, the assumptions made about the way in which the abstraction code will be invoked. These are the assumptions that the compiler for the high-level language must obey.

1.10 Structure of this thesis

The remainder of this thesis is structured as follows. Chapter 2 surveys similar and other approaches to increasing the safety of low-level and abstraction-level code. The core of the Nitro programming language is defined in chapter 3. This will be the basis from which the extensions to assist abstraction-level code will be derived. The first such extensions provide access to memory-level storage details of values. These are defined and discussed in chapter 4. Data type definitions are extensively enhanced to enable the definition of data types with specific run-time representations. This allows access to foreign values which may then be used directly by Nitro or translated into the native format of some other high-level language.

A new typing scheme is detailed in chapter 5, this infers more accurate and verbose information about values defined within a Nitro program. This typing scheme, called a *delayed type system*, is formally defined by means of a set of inference rules. These rules are used to prove important properties of the typing scheme and an algorithm is given for automatic inference of types. Finally the incorporation of the delayed typing system into the Nitro language is detailed.

The facilities to control the management of memory within a Nitro program are discussed in chapter 6.

This completes the main additions to the Nitro programming language to assist its use in abstraction-level programming. The final chapter 7, concludes with a look at the success of the additions made to Nitro, what could yet be added and also the kind of programming operations that are unlikely to be done in a functional type-safe abstraction-level programming language such as Nitro.

1.11 Contributions

The main contributions of this thesis are briefly described here. A functional abstraction-level programming language, Nitro, is developed and formally defined in chapter 3. This new language is used to investigate the application of high-level typing techniques to abstraction-level programming. To this end the language and formal definition of Nitro are extended with facilities for describing foreign data as described in chapter 4 of this thesis.

A typing scheme described in chapter 5 is developed to enhance the applicability of strong typing at a level lower than the application level. Although this was the main aim in the development of this typing scheme it is not restricted to this domain and could be applied to other kinds of programming languages. This type scheme is incorporated into the formal definition of Nitro and some properties of the typing scheme are demonstrated.

The language and formal definition are once again extended in chapter 6 to incorporate a regions memory management scheme as described in [10]. As such this thesis presents an additional experience in the use of regions to manage the life times of data objects. This increases the likelihood that this scheme will see use in practical development thereby providing more confidence in the usability of region memory management systems. The final contribution of the work done is the implementation of the nitroc compiler used to explore these ideas and available as open-source software.

1.12 Implementation of Nitro

This section describes the implementation status of the Nitro compilers. There are two versions of the Nitro compiler both developed in the functional programming language OCaml [11]. The first version was developed during the design of the foreign data interface described in chapter 4 and compiles Nitro code into assembly language which is then compiled to machine code by the host C compiler. This version is referred to as nitro_opt.

The first version was found to be very slow on larger inputs and hence a second version was developed which compiles Nitro code to C code. The second version is named nitroc. The nitroc program is more modular in design and includes three separate type checkers which can be used to type check the Nitro program. The first is the foreign type checker, this type checker is for a Hindley-Milner style type scheme augmented with type definitions for controlling how data is represented in memory, as described in chapter 4 of this thesis. The second type checker implements the delayed typing scheme described in chapter 5 of this thesis. Finally the regions type checker augments the delayed typing scheme with support for the typing of region constructs as described in chapter 6 of this thesis.

The nitroc verison of the compiler can also be used to format the input program as a document, including the responses made by the type system to top level definitions in the Nitro program. When used in this manner the comments in the Nitro program are taken to be document text. The printing engine can output in three different formats, plain text, html and latex. The latex output may then be executed to produce PostScript or PDF documents. The examples given in this thesis were produced by using the Nitro compiler in this fashion.

Henceforth the phrase "Nitro compiler" will be used to refer to nitroc. The top half of Figure 1.2 depicts the overall design of the Nitro compiler and related tools.

To compile the code into native code the Nitro compiler first converts into a private language called `miniC`. This is a subset of the C language which is informally defined by the compiler in the `miniC` module. By default all of the C types of the produced program are declared with the abstract type of `nitro_value_t`. This means that operations such as the dereferencing of a pointer value are performed using a C cast. If we trust that the Nitro type system and conversion to C have been implemented correctly then all such cast operations are safe since they correspond to safe Nitro operations. In order to increase the confidence in those implementations a type inference system for the `miniC` code produced can be invoked by the user. This system attempts to infer better types for each of the declared values in the produced C program and thus uses the (weak) C type system implemented by the C compiler to provide some assurance that no program errors were introduced by the compilation of the Nitro program.

The foreign type checker is completely separate from the two delayed typing based type checkers. It uses a Hindley-Milner style type-inference engine to infer the types. The delayed-typing type checker and the region type-checker are more closely related. The region type-checker uses the same typing engine as the delayed-typing checker but also includes inference of region types.

The top half of Figure 1.2 depicts the structure of the second version of the Nitro compiler. This shows that the three web demos depend on the front end portions of the compiler, that is the lexical analysis, parsing and type checking. In addition the common printer is required to format the results of program analysis in html. The back end portions of the compiler are not depended upon by the three web demos because no actual code is produced. By contrast the full Nitro compiler depends on both the front end and back end/code generation modules.

The bottom half of Figure 1.2 gives the flow chart of the Nitro compiler. A single run of the compiler passes the program through several stages where each stage transforms the program in preparation for the next stage of the compiler. The lexical analysis splits the program text into tokens which are more straightforward to parse. The parser then builds up an abstract representation of the program. This is a two-dimensional tree or graph representation of the one-dimensional stream of tokens produced by the lexer. From this abstract representation there is a choice of type checkers all of which produce a typed version of the abstract syntax. This is similar to the abstract syntax but decorated with the inferred type information. From this point there is a split, either the typed abstract representation may be formatted with a choice of three different output styles or the compiler will proceed to code generation. This takes the form of generation to a sub-set of the C programming language, formatting the output such that it is acceptable to a host C compiler. Optionally the mini C code which is produced may be type checked by the type checker prior to formatting for output.

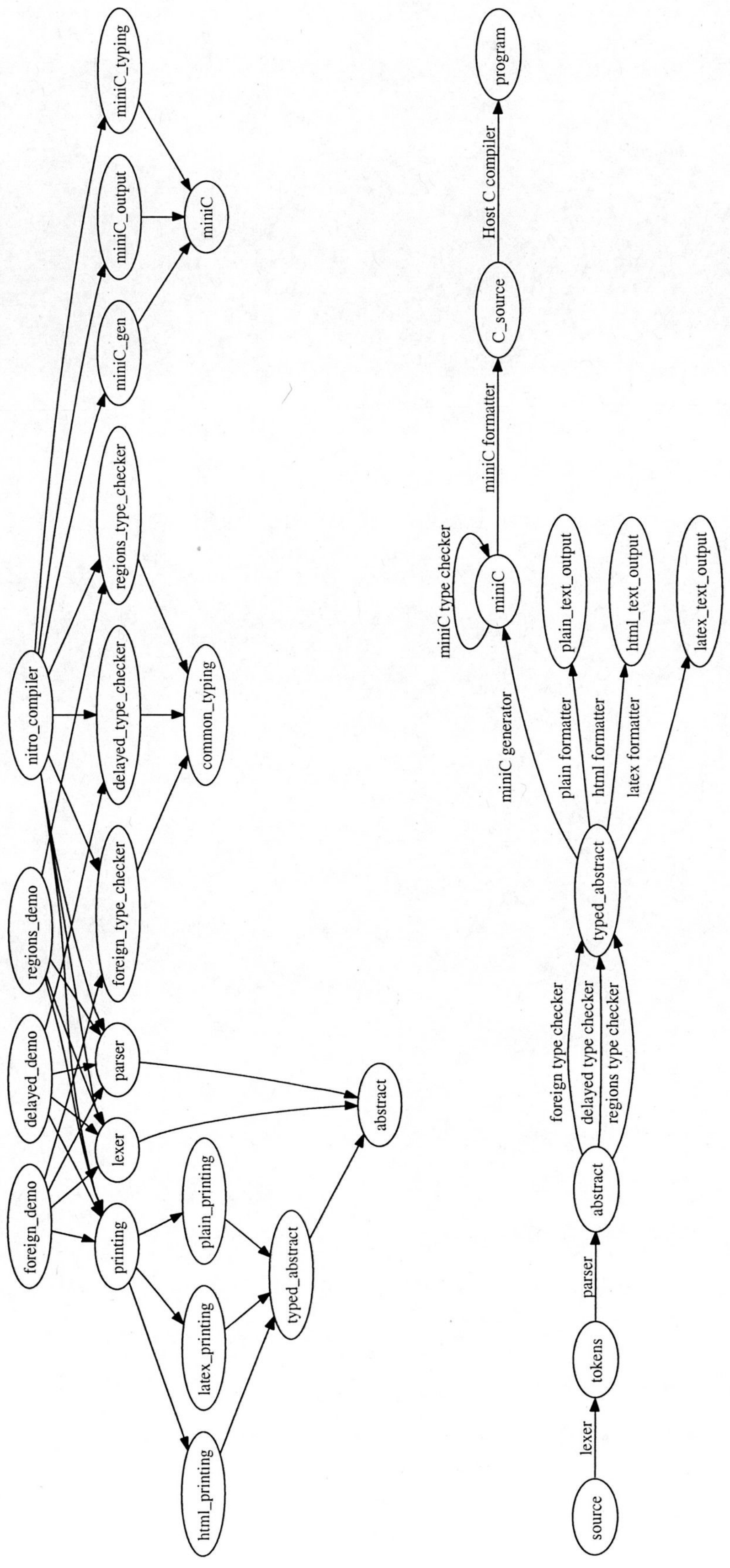


Figure 1.2: The top half depicts the structure of the Nitro compiler showing the dependencies between the separate modules. The lower half shows the flow of a single run through the Nitro compiler transforming the source text into an final executable.

Chapter 2

Background

This chapter relates some existing work relevant to the work contained within this thesis. There are three major themes; the typing of low or abstraction-level code, type systems related to that which is developed in Chapter 5 and finally work on memory management.

The first theme consists mostly of the research done in providing existing low-level programming languages with type-safety and other high-level language features, however Section 2.1 surveys research done in using existing high-level languages for low-level programming.

2.1 Functional Programming Languages

In this thesis a functional programming language is described which has been designed with abstraction-level programming as the main goal. Others have taken existing functional programming languages and applied them to the areas of systems and abstraction-level programming.

2.1.1 House

The House[12] project aims to provide access to hardware from the functional programming language Haskell via a "hardware monad". The main aim is to facilitate the implementation of an operating system fully within Haskell. The approach is to use the Haskell foreign function interface to provide access to the hardware, but restrict the use of this interface via the hardware monad. A program logic P-logic [13] is used to ensure that this interface can be used only in a safe manner. Essentially using the Haskell foreign function interface one can access the hardware but in an unsafe manner, however by accessing only through the hardware monad the user cannot make safety errors. This property is guaranteed using P-logic. The approach has proven to be fruitful, however it still relies on the unsafe marshalling routines provided by the foreign function interface, such marshalling routines could be re-written in Nitro and or the hardware monad could be ported to Nitro thus gaining still more confidence in the correctness of the low-level code.

2.1.2 The Fox Net Project

The FoxNet project is an implementation of a TCP/IP network protocol stack written entirely in an extended version of the functional programming language SML called SML+ developed for the project. The extensions provided by SML+ include a mechanism for reading and writing raw memory outside the heap managed by the SML garbage collector. The final design and implementation of the TCP/IP stack is described in [14] and the performance of the FoxNet protocol stack is reported in [15]. It was shown at the time that the implementation was significantly slower than the equivalent C implementation. This was attributed to a lack of inlining of small functions which would be implemented as C macros using `#define` in a C implementation and also the performance of the mechanism for manipulating raw memory. However the project showed that functional languages can be used for systems and low-level programming, and in addition the high-level features found in most such languages are a significant benefit when designing and implementing systems and low-level software.

2.2 The C Language and Derivatives

In this section several languages which have been derived from C are discussed as well as attempts to work within the C language to achieve a more satisfactory programming environment.

2.2.1 C++ and C#

Two extremely popular languages derived from the C programming language are C++ [16] and C# [17]. Both languages improve upon the software maintenance capabilities of C programs by using object oriented programming to allow program division. While C++ is an extension of C such that (almost) all C programs will compile with a C++ compiler, C# is an entirely new language with a similar syntax. One very large difference is that the C# language provides for automatic garbage collection. Although C# allows for the addition of low-level code using the `unsafe` modifier it is intended as a high-level application programming language and is therefore not designed for the abstraction-level programming tasks with which this thesis is concerned and is not as relevant to Nitro.

C++ however could be used by an abstraction-level programmer since it maintains the control over data representation inherited from its C origins. Coupled with this are several higher-level features, most notably the object-oriented features. In addition interfacing with a legacy C library requires no marshalling interface and the types are already a perfect match. Whilst retaining many of the advantages of low-level programming in C however, C++ also retains many of the disadvantages. In particular programming in C++ is still not type-safe. The ability to control the representation and life times of data values allows the programmer to make the same mistakes as is possible in a C implementation. The additional abstraction mechanisms provided by C++ can alleviate these problems to some extent but encapsulation mechanisms on their own cannot provide an entire solution.

Since there is greater scope for code encapsulation there is also greater scope for code reuse. This has long-understood benefits for software maintenance.

2.2.2 Cyclone

The Cyclone [18] programming language has been developed to improve the security of low-level code and in particular legacy code. This offers a very C-like syntax and semantics however the type system is more strict. In addition high-level features such as tagged-union types, polymorphism and exception raising and handling have been added. This offers a good solution for those that wish to upgrade an existing library or code incrementally into type-safe code. Rather than throw away all of the existing code it can be semi-automatically translated into Cyclone code. Additions to the code can then use the new more modern features.

In addition Cyclone has become a testing ground for new memory management techniques. This is especially true of memory management techniques related to region memory management. The region memory management scheme will be discussed further in Chapter 6 and the Cyclone language will be discussed in the conclusions chapter 7.

2.2.3 Ccured and Safe-C

The CCured[19, 20, 21] and the Safe-C projects aim to compile without modification legacy C code. However the legacy C code is compiled in such a way as to transform unsafe code into safe code. This has the significant advantage that one can take a library written in C which is desired to be ensured to be safe and compile the code without the need to understand and modify it. It may be that the initial developers of the library are not available or have no desire to improve the security of their code. The disadvantage is that because the cost of translation is never paid the dynamic cost of runtime checks and representations are forever incurred.

The CCured program analysis tool works through a combination of static analysis and inserting dynamic runtime safety checks to ensure that a C program does not make an invalid memory access. This means that some of the cost of the runtime checks can be eliminated since the cost is incurred at compile time.

The Safe-C project aims to implement a compiler which transforms unsafe C code into safe C code using the extended pointer and array access semantics described in [22], in particular this allows the efficient and immediate detection of all memory access errors.

Other similar examples of work done in increasing the trust in legacy C programs include the cqual program analysis tool[23] which allows the user to add type qualifiers to enable greater type checking of C code. Although this means that legacy code must be modified to include those extra type qualifiers it does mean that a variety of properties can be checked such as format string vulnerability [24] and deadlock detection [25].

Splint[1] is a program analysis tool aimed at detecting common C programming errors, however it does not claim to be complete in that some errors will pass through undetected.

2.2.4 Physical Type Checking

Most of the research in checking C code more strictly is based on the desire for improved security. Type systems for C whether they be static or dynamic are used to make sure that the program will not make illegal acceses into memory. Strong type systems for high-level languages aim to reduce the errors in the code by disallowing inconsistent use of program elements. For security however the correctness of the entire program is not required, only the safety, or absence of memory access errors, is important.

Research in this area has developed the concept of physical type checking of C code[26] in which two types are considered compatible if their physical representations are compatible. The basic idea is that each pointer should dereference to 'valid memory', this implies that the programmer can never consider an integer type to be a pointer. Consider the type definitions given in

Figure 2.1.

```
typedef struct point_imm_ {
  int* first ;
  int second ;
} point_imm ;

typedef struct imm_imm_ {
  int x ;
  int y ;
} imm_imm ;

typedef struct imm_point_ {
  int red ;
  int* green ;
} imm_point ;
```

Figure 2.1: Sample C type definitions.

If we have a value of type `*point_imm`, then this can be used as a value of type `*imm_imm` since when the pointer is dereferenced, the two fields can be seen as integers with no invalid pointers. However the same value cannot be seen as one of type `*imm_point` since when it is dereferenced the second field could be mistakenly dereferenced. Note that physical type checking by itself does not guarantee an absence of runtime errors such as heap management and array bounds errors.

Smith and Volpano in [27] develop a sound type system for a polymorphic dialect of C that retains many of the 'awkward' features of C such as the "address of" operator and pointer arithmetic. The type system is not intended to reject unsafe programs but give a rigorous definition of what may go wrong when a well-typed polymorphic C program is executed.

2.3 Related Foreign 'Data' Interfaces

This section surveys the work done on the foreign data interfaces for high-level languages. Any high-level language which wishes to see a high volume of users must provide a way to access libraries written in other languages since it is simply too much work to hope to port all existing (and future) libraries into your own preferred language. Most high-level languages provide some form of foreign function interface in which it is possible to call functions written in another language. A foreign data interface allows direct access to data computed by code written in a foreign language.

2.3.1 No Longer Foreign Function Interface

Nlffi [28] is an encoding of (nearly) the whole of the C type system into the type system of ML. There is a program generator which will generate some stub code required, but in general, as in this work, C values are accessed directly.

There are several advantages to using an interface in the style of Nlffi. It can be used with an existing language; there is no need to add constructs to your language to deal with foreign data. There are some cases where the Nlffi can be more efficient. These correspond exactly to those cases where C can be more efficient and in general correspond to the places where there is a possible safety hole. There is some scope for the Nlffi to be more general, again this comes from those cases where one can use runtime information to avoid some checks that a static checker must include.

The most important advantage of the Nitro foreign data interface is that we retain type safety. There is nothing in the C type system that prevents one from picking any field from a union type, regardless of what the actual value stored there is. Another example is possibly null pointers, these will be treated in Chapter 4, once represented in Nitro one cannot incorrectly attempt to dereference a null pointer. In contrast, using the Nlffi, one checks for a null pointer with `if C.Ptr.isNull l then ... else ...`. However there is nothing to prevent you dereferencing such a pointer in either branch of the conditional, or even ensuring you have performed the check. To retain type safety in this case the stub code implementing the dereferencing could apply a check, but then we would have redundantly performed the check twice.

2.3.2 Checking type safety of foreign function calls

Another approach, taken in [29], is to do multiple language type checking of the foreign function calls. The authors' goal is to check the safety of uses of an existing foreign function interface, in their case that of OCaml. This means that we still require stub code and there is still a marshalling barrier. However there is now greater confidence that the marshalling interface is indeed written correctly. The foreign data interface described in this work cannot achieve this, essentially the programmer is on their own when actually writing the interface and mistakes can still be made. A combination of the two approaches seems like a promising direction for future study.

2.3.3 Interoperability Through Common Framework

Several systems exist which aim to provide inter-operability between languages through the use of a common compilation scheme. Although such schemes have separate aims to that of Nitro they do reduce the need for marshalling routines between separate languages. However the common abstractions that such a framework provides must still be implemented. In addition it is likely that they must be implemented outwith the scheme. A common abstraction to provide is garbage collection. Such common runtimes could be re-written in Nitro.

One such framework is the Oz[30] runtime environment on top of which the Alice dialect of ML is implemented[31].

The .NET common language infrastructure[32] is another such environment aiming at providing language interoperability via a common compilation target. Although aimed mostly at object-oriented imperative languages it has been used as a target for functional languages including F#[33] and SML[34].

Additionally some languages such as SML [35, 36] have been compiled to the Java runtime thus allowing interoperability between the chosen language and Java by using the Java runtime as a common language runtime.

The Moby[37] language – a high-level statically typed language with an ML-like module language – has a framework for foreign data access [38]. The authors make a similar distinction between a foreign function interface as provided by the majority of high-level languages in common usage and a foreign data interface which allows direct access to foreign data. In addition the authors make the same observation that there are situations in which a foreign data interface is required usually due to the cost of the marshalling routines and the volume of data which must be marshalled.

The foreign data interface works by allowing the user to embed C code directly into Moby code. This allows the user very fine-grained control over the foreign data interface policy. This is done by allowing the user to include (or inline) BOL code, where BOL is the intermediate language of the Moby compiler expressive enough to be used to implement a C compiler. The disadvantage is that the BOL code itself is, as with C, weakly typed. The result is a foreign data interface more expressive than the one described for Nitro in Chapter 4 but lacking in the type safety guarantees.

2.4 Related Type Systems

In this section some of the work on type systems relevant to that of the type system developed in Chapter 5 of this thesis is discussed.

2.4.1 Lower bounds on Type Inference with Sub-typing

Hoang and Mitchell provide a proof[39] that type inference of subtypes is a P-SPACE-hard problem. This is independent of the expression syntax. This is done by showing the problem of typeability in the presence of a subtype relation to be equivalent to a safisfiability problem over a partial order. In [40] the problem is shown to have at worst an upper bound of exponential-time.

This is a somewhat negative result, however it is not the first result of its kind and would not be the first to contradict practical experience. It was generally assumed that ML type inference without the presence of a subtype relation was efficiently computable in polynomial time. However as noted in [41] various terms in ML have been constructed for which the inferred type was exponential in length. Even worse simply deciding whether or not an ML expression can be given a type is shown to be complete for exponential-time. However practical experience in the use of ML-like type inference systems has shown that most typing problems lie within those that are efficiently computable. There is therefore hope that the same may be true of type inference in the presence of subtyping though more practical experience may or may not support this hope. Experience in using the Nitro delayed typing type checker has yet to yield programs in which type checking was too slow.

2.4.2 Subtyping over Record Types

Rémy [42] improves upon the work of Wand[43] to provide record concatenation (or record extension) using row variables. This provides an intuitive extension to non-extensible record typing in an ML environment (in the author's specific case OCaml is used). Some desirable programs however are not typable under this scheme which are typable under the delayed typing scheme detailed in Chapter 5. The un-typable programs are generally due to the monomorphic restriction on function arguments. Consider the program given in Figure 2.2 written in OCaml.

```
let r = { x = true; y = 2 ; } in
   if r.x then r else { y = 1 ; }

fun r -> if r.x then r else { y = 1 ; }
```

Figure 2.2: Code showing a typable let expression and an un-typable function abstraction in Rémy's record extension of OCaml.

The system described can give a type to the first **let** expression but not the function abstraction. The delayed typing scheme can provide a useful type for both expressions.

2.4.3 Practical Type Inference for Arbitrary-Rank Types

In [44] the authors describe a practical type system for an arbitrary-rank type system. The authors note that full type inference for such a system is undecidable. However in practice software developers are prepared and often keen to write down the types of their definitions in full. This works well for the delayed typing scheme in which all top level definitions can be given a full type signature but the programmer need not be burdened with providing a type for all defined names.

2.5 Memory Management

Chapter 6 details the regions based scheme for memory management used in Nitro for programs in which the programmer wishes to control the lifetimes of their values. Many programs though can be written without regard for memory management using implicit runtime garbage collection. In Nitro the programmer has a fine degree of control over the representation of values in the program as discussed in Chapter 4. This means that it is very awkward for the compiler to provide the runtime garbage collector with the roots or pointers of the running program. Therefore the implementation depends upon a runtime garbage collector which can operate with ambiguous roots as set out in [45]. The Nitro compiler uses the realisation of this algorithm in the implementation available at: `http://www.hpl.hp.com/personal/Hans_Boehm/gc/index.html`.

In this thesis dynamic memory management is considered as one technique when in practice there are many different algorithms and combined schemes. This is not to mention the many different solutions for varied circumstances of dynamic collection including distributed, multi-processor systems and real-time environments, Wilson[46] gives a good review of available techniques for the special case of a single processor, single machine environment. Nor does this thesis begin to uncover the problem of fitting an allocator algorithm to the memory management scheme being used or the problem of allocation in general regardless of the scheme used. For a detailed look in the specific instance where memory is not expected to be moved around see [47].

Chapter 3

Core Nitro

This chapter introduces the core of the Nitro programming language. This core is a functional language lacking in any features of particular importance to the abstraction-level programmer. This will serve as the basis onto which such features will be added in subsequent chapters.

The syntax is defined in the following Section 3.1, the static semantics are defined in Section 3.2 and finally the dynamic semantics in Section 3.3.

3.1 Syntax

The syntax of Nitro will be recognised by anyone familiar with the OCaml [11] programming language. Since OCaml in turn shares some syntax with the SML programming language the syntax should be comfortable for anyone with a background in SML.

The syntax of expressions is given in Figure 3.1 and the syntax for types and type schemes is defined by the grammar in Figure 3.2. Finally the syntax of top level definitions is given in Figure 3.3.

3.1.1 Notes

On the notation used above; phrases contained with angled brackets ($<$ *phrase* $>$) are optional. Each grammar rule can be read with or without the optional

expr	$:=$	c	(constants)
	$\mid$	x	(variable access)
	$\mid$	**let** *pattern* $=$ *expr*$_1$ **in** *expr*$_2$	(let binding)
	$\mid$	(*expr*)	(bracketed)
	$\mid$	*expr*$_1$ *expr*$_2$	(application)
	$\mid$	**fun** *pattern* $\rightarrow$ *expr*	(abstraction)
	$\mid$	$\{$*field_dec+*$\}$	(record creation)
	$\mid$	*expr.field*	(field access)
	$\mid$	*expr*$_1$*.field* $\leftarrow$ *expr*$_2$	(field update)
	$\mid$	*Con expr*	(constructor application)
	$\mid$	**let rec** x $=$ **fun** *pattern* $\rightarrow$ *expr*$_1$ **in** *expr*$_2$	(recursion)
	$\mid$	**match** *expr* **with** *match* **end**	(matching)
field_dec	$:=$	*label* $=$ *expr* ;	(field initialisation)
match	$:=$	*mrule*	(one rule)
	$\mid$	*mrule* $\mid$ *match*	(many rules)
mrule	$:=$	*pattern* $\rightarrow$ *expr*	(match rule)
pattern	$:=$	_	(the any pattern)
	$\mid$	x	(identifier binding)
	$\mid$	$\{$*field_pattern+*$\}$	(record pattern)
	$\mid$	*pattern* **as** x	(named pattern)
	$\mid$	*Con pattern*	(tagged pattern)
	$\mid$	*Con*	(constructor pattern)
field_pattern	$:=$	*label* $= p$;	(field pattern)

Figure 3.1: The syntax for expressions in core Nitro.

τ	$:=$	**int**	(Integers)
	$\mid$	**bool**	(Booleans)
	$\mid$	*ident*	(Type names)
	$\mid$	*tyrow ident*	(Type application)
	$\mid$	$\tau_{arg} \rightarrow \tau_{res}$	(Functions)
	$\mid$	$'a$	(Type variables)
tyrow	$:=$	τ	(single type)
	$\mid$	(*tyrowsemi*)	(many types)
tyrowsemi	$:=$	$\tau;\tau$	
	$\mid$	$\tau;$*tyrowsemi*	

$$\begin{array}{lll} \sigma & := & \tau \\ & \mid & \forall('a_1,...'a_n).\tau \end{array}$$

Figure 3.2: The syntax for types and type schemes in core Nitro.

tydec	$:=$	**type** *tybind*	(type declaration)
tybind	$:=$	*tyvars tyname* $=$ *constrs* $\langle$**and** *tybind*$\rangle$	(tagged type dec)
	$\mid$	*tyvars tyname* $= \{$*fieldecs*$\}$ $\langle$**and** *tybind*$\rangle$	(record type dec)
tyvars	$:=$		(no type vars)
	$\mid$	$'a$	(one type var)
	$\mid$	$('a\{;'b\}^+)$	(many type vars)
constrs	$:=$	*constr* $\langle\mid$ *constrs*$\rangle$	(constructor dec list)
constr	$:=$	*Con* $\langle$*conarg*$\rangle$	(constructor dec)
conarg	$:=$	*typlace* τ	(constructor arg)
typlace	$:=$	**of**	(type placement)
fieldecs	$:=$	*fieldec*	(record field decs)
	$\mid$	*fieldec fieldecs*	(single field dec)
fieldec	$:=$	*label* : τ ;	(many field decs)

Figure 3.3: The syntax for top level definitions.

phrases. The vertical bar is used to separate grammar rules with the same left hand side but are also used by the Nitro syntax itself. The two uses are easily distinguished as the meta-notation uses are inline with the := sign of the first grammar production. The {} brackets are meta-notation except for three occurences; in the 'record creation' and 'record pattern' rules for expressions and the 'record type dec' rule for top-level declarations.

The syntax distinguishes between identifiers and tagged union type constructors. Identifiers always start with a lower case letter and constructors with an upper case letter. This means that a pattern which matches over a tagged union type cannot be mis-spelt and accidently become a catch-all identifier pattern. For example given the type definition:

type *'a option* = *Some* **of** *'a* | *None*

then the function *isNone* cannot be mis-defined as in the following definition because the compiler will complain of an undefined constructor *NONE*.

```
let isNone = fun opt →
        match opt with
            NONE → true
          | _ → false
        end
```

Match expressions require an **end** keyword as a termination symbol. If **match** expressions require no terminating symbol and are nested then it becomes ambiguous as to which match a match rule belongs to. However in contrast to SML, **let** expressions in Nitro require no terminating **end**. This means that **let** expressions can be concisely nested together.

The 'as' pattern may be confusing because several languages define this similarly but differently. In Nitro the pattern to be matched is first, followed by the **as** keyword followed by the name to which the pattern should be bound.

Recursive definitions are forced syntactically to be function expressions. This means that it is only useful to bind a single identifier rather than the more general pattern since no other pattern would match a functional value.

In SML the *ref* operator is a primitive. In contrast the OCaml and Nitro languages provide no primtive ref operator, instead both languages allow for mutable record fields. Using mutable record fields ref can be trivially defined as a record type with one mutable field:

type $'a\ ref\ = \{$**mutable** $contents : 'a\ ; \}$

The update operation may be defined as updating the $contents$ field:

let $update\_ref\ r\ x = r.contents \leftarrow x$

Since infix operators may be defined (for Nitro see the derived forms section 3.1.2) the infix update operator of SML, :=, may be defined as a synonym for $update\_ref$.

The reverse encoding of mutable record fields using a primitive ref operator is possible. However unless the compiler uses heavy optimisations the storage requirements are less compact and many operations require one extra memory access. In the next chapter Nitro's facilities for describing data layout are described and this ability to accurately define foreign updatable record fields is important. Interestingly it seems that the decision to use mutable record fields in OCaml was due to the same idea that mutable record fields subsume primitive references [48] whereas in SML the decision was made before mutable record fields were known [49] and this decision was carried forward into the later definitions.

3.1.2 Derived Forms

There are some derived forms which allow some definitions to be written out less laboriously, since these can be translated into the forms given above they need not be considered by the static and dynamic semantics defined in sections 3.2 and 3.3 respectively.

The first derived form is to allow the common $if - then - else$ style expression. This is the same as a match expression matching over the boolean type.

if e_1 **then** e_2 **else** $e_3 \Longrightarrow$

match e_1 **with true** $\rightarrow e_2 \mid$ **false** $\rightarrow e_3$ **end**

In the above syntax all function definitions must have a single identifier as the argument. In general it is more convenient to have a pattern here, particularly where the pattern is fully exhaustive. Hence we have the derived form

$\textbf{fun}\ \textit{pattern} \rightarrow e \Longrightarrow$

$\quad\quad \textbf{fun}\ x \rightarrow \textbf{match}\ x\ \textbf{with}\ \textit{pattern} \rightarrow e\ \textbf{end}$

Value declarations which involve a function definition can be translated by giving each argument following the name of the function. This works inside **let** expressions as well as **let rec** expressions. These derived forms are applied recursively until each let binding defines exactly one pattern as allowed by the core syntax described above.

$\textbf{let}\ \textit{pattern}\ \langle \textit{patterns} \rangle\ p = e \Longrightarrow$

$\quad\quad \textbf{let}\ \textit{pattern}\ \langle \textit{patterns} \rangle\ = \textbf{fun}\ p \rightarrow e$

$\textbf{let}\ \textit{pattern}\ \langle \textit{patterns} \rangle\ p\ =\ e_1\ \textbf{in}\ e_2 \Longrightarrow$

$\quad\quad \textbf{let}\ \textit{pattern}\ \langle \textit{patterns} \rangle\ =\ \textbf{fun}\ p \rightarrow e_1\ \textbf{in}\ e_2$

$\textbf{let rec}\ \textit{pattern}\ \langle \textit{patterns} \rangle\ p\ =\ e_1\ \textbf{in}\ e_2 \Longrightarrow$

$\quad\quad \textbf{let rec}\ \textit{pattern}\ \langle \textit{patterns} \rangle\ =\ \textbf{fun}\ p \rightarrow e_1\ \textbf{in}\ e_2$

Nitro does provide for the user definition of infix operators. These must all start with an operator character and the operator character is used by the parser to determine precedence. This is the same device as used in OCaml. An infix application is then made into the equivalent prefix application by the parser.

$e_1\ op\ e_2 \Longrightarrow$

$\quad\quad op\ e_1\ e_2$

Tuple expressions and patterns are translated into record expressions and patterns. Each position in such a tuple expression or pattern is given a record field label of the form *i_j* where *i* is the position in the tuple and *j* is the total length of the tuple.

This shows the derived form for the case of pairs, the rule is scaled for tuples of greater size. This same rule is applied to both patterns and expressions, so the *x* and *y* here are either both expressions or both patterns.

$(x, y) \Longrightarrow$

$\quad\quad \{1\_2 = x\ ; 2\_2 = y\ ;\}$

There are no top-level recursive value declarations. These can be simulated by pushing the recursion into a nested declaration. The rule for this is given as:

$$\begin{aligned}&\textbf{let rec}\, x = e \Longrightarrow\\ &\qquad \textbf{let}\, x = \textbf{let rec}\, x \;=\; e\, \textbf{in}\, x\end{aligned}$$

The expression to be defined recursively must be a function expression, this rule is inherited from the rule for **let rec** expressions.

An 'or-pattern' allows several patterns to be mapped to the same expression in a match rule. Often this is simply more convenient, sometimes it gives a more intuitive definition of a function. Furthermore or-patterns allow a final case in a pattern match to consist of an or-pattern which matches any of the constructors not matched above. This is more robust than a wild-card pattern since if the tagged union type in question is updated with more constructors then the compiler will emit a warning reminding the programmer to update the function definition. In addition 'or-patterns' may be efficiently compiled see [50]. For the purposes of formal definition, or-patterns are defined as a derived form. First or-patterns must be lifted to the top level of a pattern and then match rules are duplicated with respect to the top-level or-patterns.

$$\begin{aligned}&\mathit{Con}\ (p_1 \mid p_2) \Longrightarrow\\ &\qquad (\mathit{Con}\ p_1) \mid (\mathit{Con}\ p_2)\\ &\{\langle \mathit{fields}_1\rangle\ \mathit{lab} = (p_1 \mid p_2)\ ; \langle \mathit{fields}_2\rangle\} \Longrightarrow\\ &\qquad \{\langle \mathit{fields}_1\rangle\ \mathit{lab} = p_1\ ; \langle \mathit{fields}_2\rangle\} \mid \{\langle \mathit{fields}_1\rangle\ \mathit{lab} = p_2\ ; \langle \mathit{fields}_2\rangle\}\\ &(p_1 \mid p_2)\ \textbf{as}\ x \Longrightarrow\\ &\qquad p_1\ \textbf{as}\ x \mid p_2\ \textbf{as}\ x\\ &(p_1 \mid p_2) : \tau \Longrightarrow\\ &\qquad (p_1 : \tau) \mid (p_2 : \tau)\end{aligned}$$

As with the **let** binding derived form this is applied recursively until all or-patterns may be removed from the program with the following rule. $(p_1 \mid p_2) \rightarrow e \Longrightarrow$

$$p_1 \rightarrow e \mid p_2 \rightarrow e$$

Note that this means that any identifiers which are bound within an or-pattern and used in the corresponding expression must be bound in both sides of the or-pattern (and to the same type of object).

With the addition of side-effecting expressions there are some expressions which do not produce a useful result. These expressions are evaluated only for their side-effects. These expressions will often evaluate to the unit value (). Many such expressions may be sequenced together using nested **let** expressions. In addition the convenient semi-colon syntax used in both OCaml and SML is supported in Nitro. An expression may be a list of expressions separated by semi-colons. The overall result of the sequence of expressions is the result of evaluating the final expression. Such a sequence of expressions may be translated into a series of nested **let** expressions.

$e_1; e_2 \Longrightarrow$

$\quad\quad \mathbf{let}\ () \ = \ e_1\ \mathbf{in}\ e_2$

Note that this forces the type of each expression in a sequence expression (other than the final one) to be of type (). This was considered preferable to using the _ in the derived **let** expression. The programmer can always turn any other type of value into a unit value using an *ignore* function defined as:

$\mathbf{let}\ \mathit{ignore}\ x = ()$

In this way the arguably questionable 'throwing away' of a value is at least done explicitly and hence self-documented. In addition the compiler need not be very smart to remove the unnecessary matching of the unit values to the unit patterns.

The following sections formally define the static and dynamic semantics of core Nitro programs using sets of inference rules.

3.2 Static Semantics

The rules in this section define the static semantics of core Nitro. These are used to specify which programs a Nitro compiler should accept and which types should be assigned to them. Before the rules are presented the typing contexts against which program fragments are checked are described. Auxiliary functions used within the inferences rules are then detailed. The syntactic restrictions placed on Nitro programs are given in section 3.2.6 . Finally the rules for type checking begin with those for expressions in section 3.2.8.

3.2.1 Typing Contexts

Typing takes place within a typing context. A typing context contains information already known about the program. In the rules which follow C and subscripts such as C_n will be used to denote a typing context. The separate parts of a typing context are as follows.

- A *value environment*, this maps value identifiers to types. In the rules that follow *Venv* will be used to denote the value environment part of a typing context.

- A *field environment*, this maps identifiers as record field labels to the parent type in which the field is defined and the type associated with that field. In addition whether or not the field is a mutable field is stored. To denote the field environment part of a typing context, *Fenv* will be used.

- A *constructor environment*, this maps tagged type constructor identifiers to the type of the argument to the constructor and the parent tagged type within which the constructor was defined. To denote the constructor environment part of a context, *Cenv* will be used.

- A *type name environment*, this maps type names to information about types; the number of type parameters to the given type. To refer to the type name environment component of a context *Tenv* will be used.

In the semantic rules which follow the term $C + Venv$ will be used to mean the context C modified by adding the value environment *Venv* to the current value environment component of C. Also to reduce the number of rules a phrase within angled brackets $\langle\rangle$ is a *first option*. The rule can be read either with all of the first options present, or with none of the first options present. Similarly for a second option written within double angle brackets $\langle\langle\rangle\rangle$.

3.2.2 Type Schemes

A type τ_1 is said to be an instantiation of a type scheme $\sigma = \forall(('a_1 \ldots 'a_n)).\tau$ if τ_1 can be obtained by applying a substitution over the type τ such that the domain of the substitution matches the set of type variables $('a_1 \ldots 'a_n)$. We write $\tau = inst(\sigma)$ if this is the case.

Two type schemes are considered equal if they differ only in a renaming and/or reordering of the bound type variables. Also if σ_1 can be obtained by deleting the bound type variables which do not occur in the body from the type scheme σ_2 then σ_1 and σ_2 are considered equal.

3.2.3 Auxiliary Functions

There are several auxiliary functions which are used in the semantic rules for the core Nitro language. These are:

$\mathcal{CON}$ The constant type function $\mathcal{CON}$ takes as argument a literal constant and returns the type of it. For example $\mathcal{CON}(1) = \mathbf{int}$. For core Nitro the literal constants are: the integers, the boolean values (**true** and **false**) and the unit value written as ().

$\mathcal{M}$ Information about a record field label is gained from a typing context via the record field label function $\mathcal{M}$. From a given typing context and field label two types are returned. The first is the type to which the record field belongs and the second is the type associated with the particular field. In addition the field may be marked as mutable.

The two types returned may be any instantiation of the stored types with respect to the type variables stored in the type environment together with the parent record type. For a field stored as (τ_1, τ_2) where τ_1 is stored together with type variables $'a_1 \ldots 'a_n$ then the two types returned may be (τ_3, τ_4) where $\tau_3 = S(\tau_1)$ and $\tau_4 = S(\tau_2)$ for some substitution S such that the domain of S is $\{'a_1 \ldots 'a_n\}$.

This function must take special care with field names generated by tuple expressions or patterns expanded to their underlying record type form. The field label i_j contains enough information to generate both the parent tuple type and the type of the argument.

$\mathcal{R}$ Given a mapping of record-field labels to types and a typing context the record field mapping function $\mathcal{R}$ returns the parent type to which all the field labels belong.

This function only succeeds if the set of label names all belong to the same record type, all are mapped to the type defined within the definition of the parent record type, and all labels that belong to that record type are defined within the given mapping. Note that this function must also take into account the typing of tuple fields since these are not generated by type descriptions.

$\mathcal{T}$ The constructor type function $\mathcal{T}$ takes as arguments the current typing context and a constructor identifier. It returns the type of the constructor which will be an arrow type for those constructors which have an associated argument. For those constructors which have no associated argument then the type returned will simply be the parent tagged union data type which contains the constructor. Where $\mathcal{T}(C, Con) = \tau_{arg} \rightarrow \tau_{res}$, then τ_{arg} is the type of the associated argument and τ_{res} is the parent tagged union type to which the constructor *Con* belongs.

The type returned may be any instantiation of the stored type according to the type variables associated with the parent tagged union type in the same manner as described above for the $\mathcal{M}$ function.

3.2.4 Environment Closure

In order to allow the polymorphic use of identifiers defined in Nitro programs the static semantic rules will make use of a closure operation $Clos$. The closure of a type $Clos(\tau)$ is defined as $\forall('a_1 ... 'a_n).\tau$ where $'a_1 ... 'a_n$ are the type variables occurring free in τ (also written $FTV(\tau)$).

Closure can be applied according to a given semantic object, most commonly the current context so that $Clos_{C,}(\tau)$ means $\forall('a_1...'a_n).\tau$ where $'a_1...'a_n = FTV(\tau)\backslash FTV(C)$.

In addition the closure of an environment must take special care in the presence of mutable record fields. The same device employed in SML is used here. Expressions are defined as expansive or non-expansive. A non-expansive expression is defined by the grammar given in figure 3.4 and all other expressions are considered expansive.

Where the closure of a value environment *Venv* is taken, then for every identifier x in the domain of *Venv* there is a pattern p matching an expression e which defines x. The closure of *Venv* then depends on the structure of e such that where $Venv(x) = \tau$

then

$(Clos_{C,p}(Venv))(x) = \tau$

whenever e is expansive. Where e is non-expansive then as before

$(Clos_{C,p}(Venv))(x) = \forall((FTV(\tau)\backslash FTV(C)).\tau$

Note that the grammar for non-expansive expressions given could be expanded upon but it is a deliberate choice for the sake of predictability not to. For example one could allow record expressions in which all the record field initialisations are non-expansive. Having done this one could allow a field access where the record expression is non-expansive, though this kind of expression would rarely be of use. Retaining the simplicity of the grammar for non-expansive expressions and thus the ease with which a programmer can predict which expressions a compiler will allow was judged to be of greater value than allowing those extra expressions.

3.2.5 Notes

This section contains some small notes that may aid the reader in understanding these rules, particularly if the reader is familiar with similar functional languages and their definitions such as that of Standard ML[51].

Record field labels all begin with a lowercase letter and can be determined

$$
\begin{array}{rcl}
nexp & := & c \\
 & | & x \\
 & | & (nexp) \\
 & | & \textbf{fun}\ x \rightarrow e \\
 & | & Con\ \langle nexp \rangle
\end{array}
$$

Figure 3.4: Grammar of non-expansive expressions in Nitro.

as being record field labels rather than value identifiers due to their position in the source code.

All constructor identifiers and record labels belong to exactly one type. Although core Nitro does not contain a module system, were one to be added, this would provide a mechanism for the reuse of constructor identifiers and record field labels.

3.2.6 Syntactic Restrictions

In this section the syntactic restictions placed on Nitro programs are detailed. These allow the rules here to be more concise since they need not cover programs which violate the syntactic restrictions.

A pattern may not define an identifier more than once. Note that this restriction does not apply to or-patterns since these are removed by a derived-form source transformation before the syntactic restrictions are applied.

A record type declaration may not define the same record field label more than once. Similarly a record creation expression may not define a field label more than once.

Similar to record type definitions a tagged union type declaration cannot define a constructor identifier more than once.

3.2.7 Definitions

This section briefly describes the assumptions that the reader may make when reading the typing rules for expressions. These assumptions are of the typing environment in which the typing of expressions will take place and occur due to the way in which type and value definitions are treated. The treatment of type and value definitions is formalised in section 3.2.12.

A value definition will add the defined names to the value environment component of the typing context. This means that when typing an expression it can be assumed that the names of all previously defined values are in the typing context's value environment with the appropriate types.

Each record type definition defines a set of record labels which belong to the type being defined. Each record label must belong to exactly one record type. Similarly for tagged type definitions the constructor identifiers must belong to one parent tagged type. This means that when typing expressions the use of a record field label or tagged type constructor can uniquely determine the parent type involved. For example in the expression *e.label* the required type of *e* can be determined from *label*. Note that this is in slight contrast to the treatment of record labels in SML. The treatment of records in the Definition of Standard ML have been criticised by some authors for example [52].

3.2.8 Expressions

The rules in this section define the static semantics for the expressions within core Nitro. The rules all have the form $C \vdash expr \Rightarrow \tau$ where C is a typing context, *expr* is the expression to give a type to and τ is the type which may be given to the expression within the given typing context.

$\boxed{C \vdash expr \Rightarrow \tau}$

$$\frac{\mathcal{CON}(c) = \tau}{C \vdash c \Rightarrow \tau} \tag{1}$$

$$\frac{C(x) = \sigma \qquad \tau = inst(\sigma)}{C \vdash x \Rightarrow \tau} \tag{2}$$

$$\frac{C \vdash e \Rightarrow \tau}{C \vdash (e) \Rightarrow \tau} \tag{3}$$

$$\frac{C \vdash e \Rightarrow \tau}{C \vdash e : \tau \Rightarrow \tau} \tag{4}$$

$$\frac{C \vdash p \Rightarrow (Venv, \tau_p) \qquad C \vdash e_1 \Rightarrow \tau_p \qquad C + Clos_{C,p}(Venv) \vdash e_2 \Rightarrow \tau}{C \vdash \textbf{let}\ p\ =\ e_1\ \textbf{in}\ e_2 \Rightarrow \tau} \tag{5}$$

$$\frac{Venv = \{x : \tau_{arg}\} \qquad C + Venv \vdash e \Rightarrow \tau_{res}}{C \vdash \textbf{fun}\ x \rightarrow e \Rightarrow (\tau_{arg}\ \rightarrow\ \tau_{res})} \tag{6}$$

$$\frac{C \vdash e_1 \Rightarrow \tau_{arg}\ \rightarrow\ \tau_{res} \qquad C \vdash e_2 \Rightarrow \tau_{arg}}{C \vdash e_1\ e_2 \Rightarrow \tau_{res}} \tag{7}$$

The following rule allows monomorphic recursion. Notice how the closure of the environment obtained by adding the identifiers defined in the pattern is used for the typing of the body of the recursive definition, but in the intialise expression there is no closure. The effect of this is that a function cannot be used polymorphically in the body of its own definition but may be used polymorphically in the body of the let-expression just as with a non-recursive let-expression. Recall as well that due to the syntactic restrictions the intialising expression for a recursive let-binding must be a function.

$$\frac{C \vdash p \Rightarrow (Venv, \tau_p) \qquad C + Venv \vdash e_1 \Rightarrow \tau_p \qquad C + Clos_{C,p}(Venv) \vdash e_2 \Rightarrow \tau}{C \vdash \textbf{let rec}\ p\ =\ e_1\ \textbf{in}\ e_2 \Rightarrow \tau} \tag{8}$$

$$\frac{\mathcal{T}(C, Con) = (\tau_1\ \rightarrow\ \tau) \qquad C \vdash e \Rightarrow \tau_1}{C \vdash Con\ e \Rightarrow \tau} \tag{9}$$

$$\frac{\mathcal{T}(C, Con) = \tau \qquad \tau \neq (\tau_1\ \rightarrow\ \tau_2)}{C \vdash Con \Rightarrow \tau} \tag{10}$$

Match expressions (Rule 11) rely on the rules for a *match*. These are defined in section 3.2.11

$$\frac{C \vdash match \Rightarrow (\tau_1\ \rightarrow\ \tau) \qquad C \vdash e \Rightarrow \tau_1}{C \vdash \textbf{match}\ e\ \textbf{with}\ match\ \textbf{end} \Rightarrow \tau} \tag{11}$$

Note that in rule 12 the marking of the field as **mutable** or not is ignored and the application of the $\mathcal{M}$ should be regarded as allowing the return of a marked field or not. This is achieved with the use of the first option to produce two separate rules. In contrast rule 13 specifically requires that the field is marked.

$$\frac{C \vdash e \Rightarrow \tau_1 \qquad \mathcal{M}(C, label) = \langle \textbf{mutable} \rangle(\tau_1, \tau)}{C \vdash e.label \Rightarrow \tau} \tag{12}$$

$$\frac{C \vdash e_1 \Rightarrow \tau_1 \qquad \mathcal{M}(C, label) = \textbf{mutable}(\tau_1, \tau) \qquad C \vdash e_2 \Rightarrow \tau}{C \vdash e_1.label \leftarrow e_2 \Rightarrow ()} \tag{13}$$

$$\frac{C \vdash fields \Rightarrow \rho \qquad \mathcal{R}(C, \rho) = \tau}{C \vdash \{\ fields\ \} \Rightarrow \tau} \tag{14}$$

3.2.9 Record Field Initialisations

Rule 15 defines how to type a list of record field definitions of the form $label = e$; where $label$ is the record field label being defined and e is a Nitro expression. The rule here allows a list of record field definitions to be given a mapping from field names to types. Recall that the syntactic restrictions prevent the same label from being defined more than once.

$\boxed{C \vdash fields \Rightarrow \rho}$

$$\frac{C \vdash e_1 \Rightarrow \tau \qquad \langle C \vdash fields \Rightarrow \rho \rangle}{C \vdash lab = e_1\ ;\ \langle fields \rangle \Rightarrow \{lab \mapsto \tau\}\langle +\rho \rangle} \tag{15}$$

3.2.10 Patterns

The rules in this section define when a type can be given to a pattern. In order to type expressions within the scope of a pattern a Nitro compiler must also bind the identifiers defined by the pattern to a type. Therefore the rules in this section are all of the form: $C \vdash pattern \ \Rightarrow (Venv, \tau)$. Here C is the typing context, $pattern$ is the pattern to type check, $Venv$ is a mapping from the identifiers defined in *pattern* to types and τ is the type which may be given to the whole pattern.

Note that in Rule 22 in which record field labels are matched, the pattern need not define all of the labels within the given record type. The rules enforce that all of the field labels belong to the same type.

The rules begin with the underscore pattern which defines no new identifiers and thus infers the empty value environment. It may also be given any type.

$\boxed{C \vdash \mathit{pattern} \Rightarrow (\mathit{Venv}, \tau)}$

$$\frac{}{C \vdash \_ \Rightarrow (\{\}, \tau)} \tag{16}$$

$$\frac{}{C \vdash x \Rightarrow (\{x : \tau\}, \tau)} \tag{17}$$

$$\frac{C \vdash p \Rightarrow (\mathit{Venv}, \tau)}{C \vdash p \textbf{ as } x \Rightarrow (\mathit{Venv} \cup \{x : \tau\}, \tau)} \tag{18}$$

$$\frac{C \vdash p \Rightarrow (\mathit{Venv}, \tau)}{C \vdash p : \tau \Rightarrow (\mathit{Venv}, \tau)} \tag{19}$$

$$\frac{C \vdash p \Rightarrow (\mathit{Venv}, \tau_1) \qquad \mathcal{T}(C, \mathit{Con}) = (\tau_1 \rightarrow \tau)}{C \vdash \mathit{Con}\ p \Rightarrow (\mathit{Venv}, \tau)} \tag{20}$$

$$\frac{\mathcal{T}(C, \mathit{Con}) = \tau \qquad \tau \neq (\tau_1 \rightarrow \tau_2)}{C \vdash \mathit{Con} \Rightarrow (\{\}, \tau)} \tag{21}$$

$$\frac{C \vdash \mathit{fields\_patterns} \Rightarrow (\mathit{Venv}, \tau)}{C \vdash \{\mathit{fields\_patterns}\} \Rightarrow (\mathit{Venv}, \tau)} \tag{22}$$

$\boxed{C \vdash \mathit{fields\_patterns} \Rightarrow (\mathit{Venv}, \tau)}$

For the typing of a field pattern, whether or not the field is mutable is ignored. As before this is done using an optional part of the rule.

$$\frac{C \vdash p_1 \Rightarrow (Venv_1, \tau_1) \qquad \mathcal{M}(C, lab) = \langle\langle \mathbf{mutable} \rangle\rangle(\tau, \tau_1) \qquad \langle C \vdash fields\_patterns \Rightarrow (Venv', \tau) \rangle}{C \vdash lab = p_1 \,; \langle fields\_patterns \rangle \Rightarrow (Venv_1 \langle \cup Venv' \rangle, \tau)} \tag{23}$$

3.2.11 Match Rules

The rules in this section define the static semantics of a *match*. The result will be an arrow type from the common type of all the patterns to the common type of all the associated expressions.

$\boxed{C \vdash mrule \Rightarrow \tau}$

$$\frac{C \vdash p \Rightarrow (Venv, \tau) \qquad C + Venv \vdash e \Rightarrow \tau_1}{C \vdash p \rightarrow e \Rightarrow (\tau \rightarrow \tau_1)} \tag{24}$$

$\boxed{C \vdash match \Rightarrow \tau}$

$$\frac{C \vdash mrule \Rightarrow \tau \qquad \langle C \vdash match \Rightarrow \tau \rangle}{C \vdash mrule \; \langle | \; match \rangle \Rightarrow \tau} \tag{25}$$

3.2.12 Definition Semantics

Value definitions are typed as let expressions, except that there is no body expression in which the bound variables are used. Instead the bound variables are retained within the type environment and possibly used in subsequent value definitions.

$\boxed{C \vdash \mathbf{let}\; p = e \Rightarrow C_1}$

$$\frac{C \vdash p \Rightarrow (Venv, \tau) \qquad Venv_1 = Clos_{C,p}(Venv) \qquad C_1 = C + Venv_1}{C \vdash \mathbf{let}\; p = e \Rightarrow C_1} \tag{26}$$

Type definitions modify the type environment of the typing context. Record and tagged union type declarations also modify the record environment and constructor environment components of the typing context respectively.

$\boxed{C \vdash tydec \Rightarrow C_1}$

$$\frac{C_1 = C + Tenv \qquad C_1 \vdash tybind \Rightarrow (Tenv, Fenv, Cenv)}{C \vdash \textbf{type}\ tybind \Rightarrow C_1 + Fenv + Cenv} \tag{27}$$

$$\boxed{C \vdash \mathit{tybind} \Rightarrow (\mathit{Tenv}, \mathit{Fenv}, \mathit{Cenv})}$$

$$\frac{\tau = \mathit{tyname}(\mathit{tyvars}) \qquad C, \tau \vdash \mathit{fieldecs} \Rightarrow \mathit{Fenv}_1 \qquad \mathit{Tenv}_1 = \{\mathit{tyname} \mapsto \mathit{tyvars}\} \quad \langle C \vdash \mathit{tybind} \Rightarrow (\mathit{Tenv}_2, \mathit{Fenv}_2, \mathit{Cenv})\rangle}{C \vdash \mathit{tyvars}\ \mathit{tyname} \ = \ \mathit{fieldecs}\ \langle \textbf{and}\ \mathit{tybind}\rangle \Rightarrow (\mathit{Tenv}_1\langle \cup \mathit{Tenv}_2\rangle, \mathit{Fenv}_1\langle \cup \mathit{Fenv}_2\rangle, \{\}\langle \cup \mathit{Cenv}\rangle)} \tag{28}$$

$$\frac{\tau = \mathit{tyname}(\mathit{tyvars}) \qquad C, \tau, \mathit{constrs} \vdash \mathit{Cenv}_1 \Rightarrow \qquad \mathit{Tenv}_1 = \{\mathit{tyname} \mapsto \mathit{tyvars}\} \quad \langle C \vdash \mathit{tybind} \Rightarrow (\mathit{Tenv}_2, \mathit{Fenv}, \mathit{Cenv}_2)\rangle}{C \vdash \mathit{tyvars}\ \mathit{tyname} \ = \ \mathit{constrs}\ \langle \textbf{and}\ \mathit{tybind}\rangle \Rightarrow (\mathit{Tenv}_1\langle \cup \mathit{Tenv}_2\rangle, \{\}\langle \cup \mathit{Fenv}\rangle, \mathit{Cenv}_1\langle \cup \mathit{Cenv}_2\rangle)} \tag{29}$$

$$\boxed{C, \tau \vdash \mathit{fieldecs} \Rightarrow \mathit{Fenv}}$$

$$\frac{\mathit{Fenv}_1 = \{\mathit{label} \mapsto (\tau, \tau_1)\} \qquad \langle C, \tau \vdash \mathit{fieldecs} \Rightarrow \mathit{Fenv}_2\rangle}{C, \tau \vdash \mathit{label} : \tau_1 \ ; \ \langle \mathit{fieldecs}\rangle \Rightarrow \mathit{Fenv}_1\langle + \mathit{Fenv}_2\rangle} \tag{30}$$

$$\frac{\mathit{Fenv}_1 = \{\mathit{label} \mapsto \textbf{mutable}(\tau, \tau_1)\} \qquad \langle C, \tau \vdash \mathit{fieldecs} \Rightarrow \mathit{Fenv}_2\rangle}{C, \tau \vdash \textbf{mutable}\ \mathit{label} : \tau_1 \ ; \ \langle \mathit{fieldecs}\rangle \Rightarrow \mathit{Fenv}_1\langle + \mathit{Fenv}_2\rangle} \tag{31}$$

$$\boxed{C, \tau \vdash \mathit{constrs} \Rightarrow \mathit{Cenv}_1}$$

$$\frac{\mathit{Cenv}_1 = \{\mathit{Con} \mapsto (\tau_1 \ \rightarrow \ \tau)\} \qquad \langle C, \tau \vdash \mathit{constrs} \Rightarrow \mathit{Cenv}_2\rangle}{C, \tau \vdash \mathit{Con}\ \textbf{of}\ \tau_1\ \langle |\ \mathit{constrs}\rangle \Rightarrow \mathit{Cenv}_1\langle + \mathit{Cenv}_2\rangle} \tag{32}$$

$$\frac{\mathit{Cenv}_1 = \{\mathit{Con} \mapsto \tau\} \qquad \langle C, \tau \vdash \mathit{constrs} \Rightarrow \mathit{Cenv}_2\rangle}{C, \tau \vdash \mathit{Con}\ \langle |\ \mathit{constrs}\rangle \Rightarrow \mathit{Cenv}_1\langle + \mathit{Cenv}_2\rangle} \tag{33}$$

3.3 Dynamic Semantics

This section presents the dynamic semantics of core Nitro programs. An expression may evaluate to either a value, as described in the following section, or the special value *STOP*. Since core Nitro does not have an exception mechanism *STOP* is a fatal error. This means that the program will stop running, and the implementation is free to choose whether or not to display some diagnostic message to the user. The *STOP* value does not indicate that a type error occurred.

A pattern is evaluated against a specific value and will return either a new value environment or *FAIL* to indicate that the value did not match the pattern. Note that a *FAIL* does not indicate a fatal error since there may be further patterns to try. Once all patterns have been exhausted a *FAIL* is promoted to a *STOP*.

Only well-typed programs are considered. This simplifies the rules somewhat and also means that there are specific circumstances under which *STOP* will occur. The only situation is a pattern failure where there are no more patterns to try. A simple example is the following let-expression:

let *Some x* = *None* **in** *x*

3.3.1 Representation of Values

A value may be as described in the following table. The first column in this table gives the identifiers which will be used in the following rules to denote a member of the given set. The next column is the name of the set. The third column describes the contents of the set.

v	$\in$	*Value*	$=$	$Record \cup Tagged \cup FcnClosure \cup Constant$
r	$\in$	*Record*	$=$	$Lab \overrightarrow{fin} Val$
t	$\in$	*Tagged*	$=$	$Con \cup Con \times Val$
(x,e,E,VE)	$\in$	*FcnClosure*	$=$	$ValueId \times Exp \times Env \times VEnv$
c	$\in$	*Constant*	$=$	predefined constants
a	$\in$	*Address*	$=$	memory locations

3.3.2 Function Closures

Function closures are the representation of functional values. These match closely the representation of functional values given in [51]. The informal definition of a function closure (x, e, E, VE) is that the environment E holds the environment at the time of the function creation. The value environment VE will hold the recursive functions.

3.3.3 The Rec operation on value environments

The *Rec* function operates on value environments and is intended to perform a single unrolling of recursively defined functions ready for their possible application. The recursive function must have the following properties:

- $Dom(VE) = Dom(Rec(VE))$
- if $VE(x) \notin FcnClosure$ then $VE(x) = (Rec(VE))(x)$
- if $VE(x) = (y, e, E, VE_2)$ then $(Rec(VE))(x) = (y, e, E, VE)$

As mentioned the recursive functions which are contained within the VE component of a function closure are unrolled once at definition time and once again at application time. This means each time it is applied recursively it is unrolled once. The *Rec* function is therefore applied in two rules: Rule 37 for application and Rule 40 for recursive declaration. An example is shown in the notes section 3.3.9.

3.3.4 The State and $STOP$ Conventions

The state consists of a mapping from addresses to values. The dynamic semantics take place within a global state which is updated and inspected only by those rules which correspond to the record-related expressions. However all other rules which contain sub-expressions must observe the possible side effects of the sub-expressions. This can be laborious and detract from the main purpose of each rule, hence the rules which follow which do not explicitly mention the state are subject to a convention which states that if a rule is presented as:

$$\frac{E \vdash phrase_1 \Rightarrow v_1 \qquad \ldots \qquad E \vdash phrase_n \Rightarrow v_n}{E \vdash phrase \Rightarrow v}$$

Then this rule is expanded to mean:

$$\frac{E, s_0 \vdash phrase_1 \Rightarrow v_1, s_1 \qquad \ldots \qquad E, s_{n-1} \vdash phrase_n \Rightarrow v_n, s_n}{E, s_0 \vdash phrase \Rightarrow v, s_n}$$

In this way the side-effects of each of the sub-expressions of the main phrase are accumulated and inherited by the conclusion for the whole rule. In addition the order of evaluation is set to be left to right according to the order of the premises. This convention will be referred to as the state convention.

The *STOP* convention is similar. An expression may produce *STOP* if any of its sub-expressions do so. In this instance however we require multiple rules for each sub-expression which may fail. One can infer an additional rule for each sub-expression, such that all sub-expressions to the left have not failed, the current sub-expression fails and the conclusion is that the whole expression fails.

If a rule is presented as:

$$\frac{E \vdash phrase_1 \Rightarrow v_1 \qquad \ldots \qquad E \vdash phrase_n \Rightarrow v_n}{E \vdash phrase \Rightarrow v}$$

then the first additional rule is:

$$\frac{E \vdash phrase_1 \Rightarrow STOP}{E \vdash phrase \Rightarrow STOP}$$

and the others are all of the form:

$$\frac{E \vdash phrase_1 \Rightarrow v_1 \qquad \ldots \qquad E \vdash phrase_{i-1} \Rightarrow v_{i-1} \qquad E \vdash phrase_i \Rightarrow STOP}{E \vdash phrase \Rightarrow STOP}$$

As a small note it should be observed that the expansion of the stop convention may cause the same rule to be generated twice. Such duplicate rules can be safely ignored. As an example the rules for match expressions 43 and

44 both expand to give the same rule indicating a $STOP$ value is the result in the case that the first sub-expression fails.

3.3.5 Expressions

$\boxed{E \vdash expr \Rightarrow v/STOP}$

$$\frac{val(c) = v}{E \vdash c \Rightarrow v} \tag{34}$$

$$\frac{E(x) = v}{E \vdash x \Rightarrow v} \tag{35}$$

$$\frac{}{E \vdash \textbf{fun}\; x \rightarrow e \Rightarrow (x, e, E, \{\}) \text{ in } FcnClosure} \tag{36}$$

$$\frac{\begin{array}{cc} E \vdash e_1 \Rightarrow (x, e, E_1, VE) \text{ in } FcnClosure & E \vdash e_2 \Rightarrow v_2 \\ E_2 = E_1 + \{x \mapsto v_2\} + Rec(VE) & E_2 \vdash e \Rightarrow v \end{array}}{E \vdash e_1\; e_2 \Rightarrow v} \tag{37}$$

$$\frac{E \vdash e_1 \Rightarrow v_1 \qquad E, v_1 \vdash p \Rightarrow VE \qquad E + VE \vdash e_2 \Rightarrow v}{E \vdash \textbf{let}\; p \;=\; e_1 \;\textbf{in}\; e_2 \Rightarrow v} \tag{38}$$

$$\frac{E \vdash e_1 \Rightarrow v_1 \qquad E, v_1 \vdash p \Rightarrow FAIL}{E \vdash \textbf{let}\; p \;=\; e_1 \;\textbf{in}\; e_2 \Rightarrow STOP} \tag{39}$$

Note that for recursive let declarations the syntactic restrictions force the intialising expression to be a function. Since the only pattern kinds which can have function type cannot fail then the program would not pass type checking if the pattern could fail. For this reason there is no rule equivalent to rule 39.

$$\frac{E \vdash e_1 \Rightarrow v_1 \qquad E, v_1 \vdash p \Rightarrow VE \qquad E + Rec(VE) \vdash e_2 \Rightarrow v}{E \vdash \textbf{let rec}\; p \;=\; e_1 \;\textbf{in}\; e_2 \Rightarrow v} \tag{40}$$

$$\frac{}{E \vdash Con \Rightarrow Con \text{ in } Value} \tag{41}$$

$$\frac{E \vdash e \Rightarrow v}{E \vdash Con\ e \Rightarrow (Con, v) \text{ in } Value} \tag{42}$$

$$\frac{E \vdash e \Rightarrow v \qquad E, v \vdash match \Rightarrow v_1}{E \vdash \textbf{match}\ e\ \textbf{with}\ match\ \textbf{end} \Rightarrow v_1} \tag{43}$$

$$\frac{E \vdash e \Rightarrow v \qquad E, v \vdash match \Rightarrow FAIL}{E \vdash \textbf{match}\ e\ \textbf{with}\ match\ \textbf{end} \Rightarrow STOP} \tag{44}$$

Rule 44 covers the case where all the patterns in the match rules of a match expression fail to match the value. The dynamic semantics of *match* are defined in section 3.3.6.

3.3.5.1 Records

This section defines the dynamic semantics of record expressions. This includes record creation, record field access and record field update. These rules are distinct from the rules defined above in that they explicitly mention the state and hence are not subject to the state convention.

$$\frac{s, E \vdash fields \Rightarrow r \text{ in } Value, s' \qquad a \notin Dom(s')}{s, E \vdash \{fields\} \Rightarrow a, s' + \{a \mapsto r\}} \tag{45}$$

$$\frac{s, E \vdash e \Rightarrow a \in Addr, s' \qquad s'(a) = r \text{ in } Record \qquad r(label) = v}{s, E \vdash e.label \Rightarrow v, s'} \tag{46}$$

$$\frac{\begin{array}{c} s, E \vdash e_1 \Rightarrow a \text{ in } Addr, s_1 \qquad s_1, E \vdash e_2 \Rightarrow v, s_2 \qquad s_2(a) = r \text{ in } Record \\ r_2 = r \backslash label + \{label \mapsto v\} \qquad s_3 = s_2 \backslash \{a\} + \{a \mapsto r_2\} \end{array}}{s, E \vdash e_1.label \leftarrow e_2 \Rightarrow (), s_3} \tag{47}$$

3.3.5.2 Field Initialisations

$\boxed{E \vdash fields \Rightarrow r/STOP}$

$$\frac{E \vdash e \Rightarrow v \qquad \langle E \vdash fields \Rightarrow r \rangle}{E \vdash label = e \; ; \; \langle fields \rangle \Rightarrow \{label \mapsto v\} \langle + r \rangle} \tag{48}$$

3.3.6 Matches

A match is considered against a given value. For match rules it must be distinguished between the pattern failing and the associated expression evaluating to *STOP*. This is the reason that *STOP* is distinct from *FAIL*. In the case that the pattern fails to match the given value then *FAIL* is produced and in Rule 50 if there are more match rules to follow then these are tried.

By the *STOP* convention Rule 49 states that if the first match rule produces a *STOP* then the whole match also produces a *STOP*.

In summary there are three cases to consider.

- The pattern of the first (and perhaps last) match rule matches the given value. In this case the the result is either a new value or *STOP* depending on what the expression of the first match rule evaluates to.
- The pattern of the last match rules fails to match the value. This in turn implies that no pattern in any of the match rules have matched the given value otherwise this pattern would not have been tried. In this case the whole match results in a *FAIL*. This case is covered by rule 51.
- The pattern of the first match rules fails in which case the subsequent match rules are considered against the given value.

$\boxed{E, v \vdash match \Rightarrow v_1 / FAIL / STOP}$

$$\frac{E, v \vdash mrule \Rightarrow v_1}{E, v \vdash mrule \; \langle | \; match \rangle \Rightarrow v_1} \tag{49}$$

$$\frac{E, v \vdash mrule \Rightarrow FAIL \qquad E, v \vdash match \Rightarrow v_1 / FAIL}{E, v \vdash mrule \mid match \Rightarrow v_1 / FAIL} \tag{50}$$

A single match rule then can either produce *FAIL* if the pattern fails to match the given expression. If the pattern does match the value then either a result value or *STOP* is produced based on what the associated expression evaluates to.

$\boxed{E, v \vdash mrule \Rightarrow v_1/FAIL/STOP}$

$$\frac{E, v \vdash p \Rightarrow FAIL}{E, v \vdash p \rightarrow e \Rightarrow FAIL} \tag{51}$$

$$\frac{E, v \vdash p \Rightarrow VE \qquad E + VE \vdash e \Rightarrow v_1}{E, v \vdash p \rightarrow e \Rightarrow v_1} \tag{52}$$

3.3.7 Patterns

Patterns are matched against a given value and produce a value environment or *FAIL* if the pattern does not match the given value.

$\boxed{E, v \vdash pattern \Rightarrow VE/FAIL}$

$$\frac{}{E, v \vdash \_ \Rightarrow \{\}} \tag{53}$$

$$\frac{val(c) = v}{E, v \vdash c \Rightarrow \{\}} \tag{54}$$

$$\frac{val(c) \neq v}{E, v \vdash c \Rightarrow FAIL} \tag{55}$$

$$\frac{}{E, v \vdash x \Rightarrow \{x \mapsto v\}} \tag{56}$$

$$\frac{v = Con}{E, v \vdash Con \Rightarrow \{\}} \tag{57}$$

$$\frac{v \neq Con}{E, v \vdash Con \Rightarrow FAIL} \tag{58}$$

$$\frac{v = (Con, v_1) \qquad E, v_1 \vdash p \Rightarrow VE/FAIL}{E, v \vdash Con\ p \Rightarrow VE/FAIL} \tag{59}$$

$$\frac{v \neq (Con, v_1)}{E, v \vdash Con\ p \Rightarrow FAIL} \tag{60}$$

$$\frac{v = a \text{ in } Addr \qquad s(a) = r \text{ in } Value \qquad E, r \vdash fields \Rightarrow VE/FAIL}{s, E, v \vdash \{fields\} \Rightarrow VE/FAIL, s} \tag{61}$$

Note that in Rule 61 the state is mentioned – as it must be examined – but it is not modified.

3.3.8 Field Patterns

$\boxed{E, r \vdash fields\_patterns \Rightarrow VE/FAIL}$

$$\frac{r(lab) = v \qquad E, v \vdash p \Rightarrow FAIL}{E, r \vdash lab = p\ ; \langle fields\_patterns \rangle \Rightarrow FAIL} \tag{62}$$

$$\frac{r(lab) = v \qquad E, v \vdash p \Rightarrow VE \qquad \langle E, r \vdash fields\_patterns \Rightarrow VE_1/FAIL \rangle}{E, r \vdash lab = p\ ; \langle fields\_patterns \rangle \Rightarrow VE \langle + VE_1/FAIL \rangle} \tag{63}$$

3.3.9 Notes

The use of the *Rec* function can now be shown via an example. Consider the definition of a recursive function such as:

let rec $count\ x = count\ (x+1)$

recall that this is a derived form for:

let $count =$ **let rec** $count\ =$ **fun** $x \longrightarrow count\ (x+1)$ **in** $count$

So by the rule for abstraction (Rule 36) the abstraction expression evaluates to

$(x, count\ (x+1), E, \{\})$ in $FcnClosure$

due to the use of *Rec* in the rule for recursive let bindings (Rule 40) this becomes:

$(x, count\ (x+1), E, \{count \mapsto (x, count\ (x+1), E, \{\})\})$

So in other words the mapping of the recursive function *count* contains a single mapping of the *count* function itself unrolled once. Whenever the *count* function is applied, when the expression component of the function closure itself is evaluated, this 'one unrolling' occurs. Therefore the *count* function is always mapped in the environment containing a finite number of unrollings of the function body.

Chapter 4

Foreign Data Interface

This chapter details the facilities provided in Nitro for inspecting and creating values which must be manipulated by code written in another programming language. The other language may be a lower-level language such as C or a higher-level language such as Java. The main purpose of this is to allow interfacing with other languages.

4.1 Motivation

A language allows the user to manipulate data through the use of primitives. Primitives allow the creation, inspection and manipulation of data. The primitives in a low-level language allow the user to directly access the representation of data in the user's program. A high-level language does not allow the user to directly inspect the representation of data in the user's program. The primitives provided by the high-level language are such that a language implementation may change the underlying representation of data and all uses of the given primitives are still valid. Therefore the data inspection and manipulation primitives can be seen as an opaque interface. This allows an implementation of the high-level language to ensure that all of the values in a program conform to a particular common representation. This in turn allows the implementation to provide other runtime services such as garbage collection.

The programmer cannot directly use data from another language, because that data will not conform to the relevant internal representation for this language. The garbage collector and other runtime services are usually written in another language, because it manipulates that internal representation which we are not allowed to inspect.

An abstraction-level programming language must provide the ability for the programmer to control the representation of data in a program once it is run. The principal reason for this is to enable the abstraction-level programmer to inter-operate with the values of the high-level language for which the abstraction is being provided. In order to provide an abstraction the language may also be required to interface with operating system constructs or even directly with the hardware. Further advantages of providing low-level control over data representation in the abstraction-level language include the ability to optimise abstraction routines such that the performance loss in utilising the abstraction is as slight as possible.

This chapter will introduce the foreign data interface of Nitro. The programmer is given the ability to describe how values of specific types are represented in memory at runtime. This allows the programmer to access foreign data. The ability the programmer has to define the layout of data structures can also be used for data private to Nitro code. That is, code which does not interface with the outside world. This means that the programmer can optimise data structures that are not foreign or exported. For this to be a useful addition we insist that the programmer is still unable to subvert the type system of Nitro. The programmer should therefore be unable to access a value as a value of an incompatible (Nitro) type. The type system is a static type system, although some dynamic checks must be inserted such as array bounds checks.

4.1.1 Marshalling

A marshalling interface is a common way for a high-level language to provide access to foreign data. Because the high-level language only permits operations on data which corresponds to its representation of values, the language must provide a means to marshal arbitrary values to and from that internal representation. Furthermore because the language does not provide the ability to inspect arbitrary values, the marshalling code must be written in a language which does. In other words the marshalling code must be provided by an abstraction-level programming language. Usually an implementation of such a high-level language will provide C macros and functions allowing the creation of values conforming to the internal representation. An interface is then built from one language to another by using C to inspect the values of the first language and create values of the second using the provided functions. Unfortunately this means that marshalling routines must be written for each separate implementation of the language. In practice this is often offset by the language designer specifying a common set of C macros and procedures which must be provided by an implementation. Alternatively there exist several attempts to generate C stub code automatically, see for example [53, 54].

For an abstraction-level programming language this is not suitable. Marshalling routines are exactly the kind of programming task that an abstraction-level programming language is used for and it is not appropriate to require the use of a further low-level language to manipulate foreign data structures. An alternative arrangement is to provide our abstraction-level language with the facilities to inspect and manipulate foreign values in a way that we can ensure is safe. To do this we provide the ability to describe the representation of a foreign data type. A programmer may still write an unsafe program, if they make a mistake when writing the interface. If however, the interface is correct, then the type-system of the language can make sure that no illegal operations over the foreign data are performed. Additionally the compiler can help detect an illogical interface, such as an ambiguous union type where a value could be considered to be of two incompatible types because the test cannot always determine between the two.

In addition because Nitro is closer to a high-level language than C or other low-level languages, Nitro is a good candidate for writing new applications that must interface efficiently with legacy code. Suppose there is a library written in a low-level language such as C, which provides some important functionality required by a new application, for example the parsing of XML[55] data. If this new application is written in a high-level language then marshalling routines must be written to convert the C representation of the parsed XML data into the representation of the high-level language. It may be the case that the application requires many calls to the C parsing library and the performance loss due to the repeated calls to the marshalling routines is too expensive. In such cases the high-level language is often abandoned and the new application written in the unsuitable low-level language C. In these cases Nitro, an abstraction-level programming language, can be used as a compromise. Nitro can directly access the C representation of the parsed XML data and so no marshalling routines are required. However the purpose of Nitro is to provide the abstraction-level programmer with desirable high-level language features, and is hence a more suitable language in which to implement the remainder of the new application than the low-level language C.

4.2 Data Representation Facilities

This section describes the various facilities provided by Nitro for using tagged types to describe external types. Each new facility is described in a subsection, which ends with a discussion of the typing requirements introduced by the addition of the associated extension. This section aims to give an informal account of the data representation constructions. The syntax and semantics, both static and dynamic, are formalised in Sections 4.3, 4.4 and 4.5. To begin with the basic idiom of a tagging representation of data is reviewed.

4.2.1 Tags in data representation

A common C idiom is a tagged union type. Union types are used as the type of a storage location into which values of two or more types can be stored. When we wish to use the value stored in a location of union type, we cast the storage location to the type that we expect the value stored there to be. Often we use a tag or kind field to record what type is stored within the union location. Here is an example:

```
typedef int circle_dimensions;
typedef int square_dimensions;
typedef struct rect_d
{ int width; int height; }
  rect_dimensions;

typedef enum { Circle = 1,
               Square = 2,
               Rectangle = 3
} shape_kind;

typedef struct shape_ {
  shape_kind tag;
  union { circle_dimensions circle;
          square_dimensions square;
          rect_dimensions rectangle; } dimensions;
} shape;
```

The intention is that a value of type **struct** shape_, holds a tag indicating how the rest of the value should be interpreted. So that the value of the tag field corresponds with the type of the value stored within the **union** dimensions field. Often the programmer would then write macros for creating and accessing valid shape values in a consistent manner. A high-level language will often provide a special syntax for defining such tagged types and creating and inspecting tagged values. Additionally, because it is not possible to examine the underlying representation of such a tagged value, it is not possible to subvert the type system and access or create a value without using the tag. Therefore all manipulations can be checked and ensured to be sensible.

Generally the lack of control over the underlying representation of tagged values is of no concern. It does not matter to us whether the tagged value is represented with the tag first and then the value, or vice versa. Also it does not matter what numerical values are given to each of the tags. It might be defined by something like:

```
type shape = Circle of int
           | Square of int
           | Rectangle of (int, int)
```

However, where the type is that of a foreign language, such as the C `shape` type defined above, we cannot use such a type, since it is unlikely to be represented in the same way. If we could define where to place the tag and which numerical values each tag should take, then we would be able to define the C `shape` type in our high-level language. We could access such a value directly from Nitro without marshalling. Also the type system of Nitro ensures that we cannot create an invalid `shape` value. For example, where the union field is set to a value of type `circle_dimensions` but the `tag` field is set to `Rectangle`. It also ensures that we do not make a mistake when interpreting the contents of a `shape` value.

4.2.2 Custom tags

In the previous section we wanted to use the constructors of a tagged type to represent the enumeration type used in the C code. We were prevented from doing this because our tags would take on numerical values determined by the Nitro compiler. Hence our first extension to the language of tagged types is to allow the programmer to specify the value of each constructor's tag. We do this using braces after the constructor name. With this extension we can implement the enumeration type used in the `shape` type of the previous section.

```
type shape =
  Circle {1}
| Square {2}
| Rectangle {3}
```

Note that this will not allow us to subscript the union `dimensions` field of the C struct. For that we require to add arguments to our Nitro `shape` type. A value of this type will consist of a pointer to a word in memory which contains one of the tag values.

4.2.2.1 Typing

There is no additional typing required for this extension. One might worry about the possibility of assigning the same tag to two distinct constructors. This does not affect type safety at this stage, however it will affect type safety when we consider arguments to custom tagged types. In addition it is sometimes desirable for two constructor identifiers to be interchangable, for example we may define a colour enumeration type which includes the two constructors:

```
| Grey {1}
| Gray {1}
```

4.2.3 Arguments

The example which provoked our use of tagged union types was the common C idiom of using a value of an enumeration type to distinguish between the different possibilities stored in a location of union type. In general we want to use a tag to record the type of the values that surround it in memory. We state the type of the argument after the value of the tag in the type definition. The **precedes** keyword is used to indicate that the argument is stored after the tag in memory. When we match against a tag with a **precedes** argument, the pointer to the tag is incremented to give a value of the type of the argument, which must therefore be a pointer type. This allows us to describe several locations after the tag by using the tuple type.

Single ground type arguments cannot be given with the **precedes** keyword because a ground type is not a pointer type. For example the argument **precedes int** does not make sense and will be rejected by the compiler. One can of course still use the **of** keyword to indicate ground type arguments which follow a tag.

With this additional facility we have all the tools necessary to fully describe our C `shape` type. The definition is:

```
type shape =
  Circle {1} of int
| Square {2} of int
| Rectangle {3} precedes (int, int)
```

4.2.3.1 Typing

The type checker must now worry about the possibility of two tags having the same value. Clearly if we accidently defined the `Rectangle` constructor above as having tag 2, then we could match against it when the real value was created with the `Square` constructor. This would mean we would incorrectly assume that two integers followed the tag, when in fact there is only one. There is more than one solution to this problem. The simplest is to reject any type definition for which two constructors with identical tags do not have identical argument types. Other possibilities are of little use at this point and are considered later. The final criteria is formalised in the static semantics in Section 4.4.

4.2.4 Unboxed Values

A further requirement for enabling the inspection of foreign values is that we are able to match long values without the dereferencing of a pointer as required above with the simple custom tagged type. To this end Nitro provides the **`immediate`** keyword to represent an unboxed value. Enumerated types become more efficient, since we need not allocate space on creation of a value and we also do not incur the cost of the dereference when matching against the value. In fact we can now interface perfectly with C enumerated types. We do not want to use an immediate type for the above `shape` type though, because the C type that we are interfacing with is not an immediate enumerated type, it is a struct type containing an immediate enumerated type.

The **immediate** type will be expanded upon later to cover more situations that straightforward enumeration types. This will include the introduction of tags which do not take up all of the word in which they are stored, hence arguments to an **immediate** type can be stored in the word of the tag itself. This is a common space optimisation in legacy C code. Also we later allow

```
struct tm
{
  int tm_sec;/* Seconds.[0-60] (1 leap second) */
  int tm_min;/* Minutes.[0-59] */
  int tm_hour;/* Hours.[0-23] */
  int tm_mday;/* Day.[1-31] */
  int tm_mon;/* Month.[0-11] */
  int tm_year;/* Year- 1900.  */
  int tm_wday;/* Day of week.[0-6] */
  int tm_yday;/* Days in year.[0-365]*/
  int tm_isdst;/* DST.[-1/0/1]*/

#ifdef__USE_BSD
  long int tm_gmtoff;/* Seconds east of UTC.  */
  __const char *tm_zone;/* Timezone abbreviation.  */
#else
  long int __tm_gmtoff;/* Seconds east of UTC.  */
  __const char *__tm_zone;/* Timezone abbreviation.  */
#endif
} ;
```

Figure 4.1: C code defining the tm struct in the time.h header file.

tags to overlap. This means that we can interface with possibly-null pointers in an elegant way.

Here we define the days of the week type, that could be used to represent the tm_wday field of a tm struct as defined in the time.h system header file (on Posix compliant systems[56]) and shown in Figure 4.1. Notice that although the C code expects an integer our type would be more robust since we ensure that we cannot create an invalid weekday value.

```
type immediate weekday =
  Sunday {0} | Monday {1} | Tuesday {2} | Wednesday {3}
| Thursday {4} | Friday {5} | Saturday {6}
```

4.2.4.1 Typing

There are no special requirements on the type checker for immediate types at this point. However, the extensions mentioned above which will be discussed in the sections which follow do require modification to the type system.

4.2.5 Masks for custom constructors

Often the number of different tags that a value may have is quite limited. The arguments of the tags can also have quite small space requirements. When this is the case there is always the temptation to pack the tags and the values together into a single word. This is particularly the case with legacy code where - at the time the code was written - storage space was at a premium. If we want to interface with such code then we must provide a way for a tag to match against only a portion of the tag value.

As an example we consider how a garbage collector may tag values. A garbage collector must decide at runtime whether a value represents a pointer into memory or is simply a long value representing an integer or a value of some enumerated type. A common way to do this is to reserve the least significant bit of every value to tag the value as a pointer or an integer. Pointers are generally word aligned and hence distinguished by a zero in the least significant bit, whereas integers are marked with a one in the least significant bit, and must be shifted appropriately when loading and storing from memory. The main implementation of the OCaml language is an example of a language which uses an internal representation such as this.

To recap, we need to match values against tags but only a portion of the value is matched against the tag. We require a type which reflects this. In Nitro we can use the **mask** keyword to modify a tag value. This means that when matching against a tag, the candidate tag value is first masked before being compared with the defined tag value. Here is a type for OCaml values.

```
type immediate ocaml_value =
   Ptr  { 0 with mask 1 } of heap_object
 | Long { 1 with mask 1 } of int
```

```
let print_ocaml_value v =
   match v with
     Ptr hp -> print_heap_object hp
   | Long i -> print_int (i >> 1)

let create_long i =
  Long ((i << 1) + 1)
```

heap_object type and a function print_heap_object to print values of this type. We have used the **immediate** keyword to operate on values as they are encountered without interpreting them as pointers to tags. Our function, print_ocaml_value, matches against either tag which only takes up part of the value. Hence the argument here refers to the whole value. Our **immediate** tags before had no argument since this did not make sense; if you match against the whole value then you know the whole value and do not need an argument to represent it. However when we match against only a portion of the tag value this usually means that the rest of the tag value, o[Br all of it, represents some value.

We have not yet described how to use only a part of the tag value as the argument, so in this example the whole tag value is also the argument value. As described above this is perfect for pointer values since their representation does not need to be changed. For long values this means that we have to remember to shift the argument. In a later section we will see how this can be done for us automatically.

4.2.5.1 Failing constructors

Adding arguments to an **immediate** tagged value changes the way that we construct a value of that type. When the tag uses up all of the space of the tag value, then using that constructor is equivalent to using a constant. The operation cannot fail. However when we accept an argument, the argument may not be of the correct form, that is, it may not match the tag. Hence creating values with an **immediate** constructor that accepts an argument is an operation that may fail. In this case the computation fails and exits. When Nitro introduces exceptions a pre-defined exception may instead be raised. Where == refers to physical equality the pseudo code for the constructor application expression, (*Con v*) then becomes:

```
let v' = Con v in
  let Con v'' = v' in
  if v'' == v
  then v'
  else error
```

4.2.5.2 Typing

As stated earlier, allowing tag values to be explicitly defined opens up the possibility of tags being equal. When constructors may have arguments this causes a type safety hole. Now that tags are permitted to match only a part of the value there is greater scope for two tags to overlap and also for overlapping tags to be useful.

Suppose we wish to add to our OCaml value type the possibility for a null pointer. This might be useful if we wished to use the same internal representation to compile a language which allows null pointers. We might then try to extend our definition like this:

```
type immediate ocaml_value =
    Null { 0 }
  | Ptr  { 0 with mask 1 } of heap_object
  | Long { 1 with mask 1 } of int
```

Without due care from the type system this could create a safety hole. The `Ptr` tag could match a value that had been created with the `Null` constructor. If we then tried to interpret this as a `heap_object` we would then make an invalid memory access.

In fact it is likely that we would have written many functions that would now be rendered unsafe and need to be updated. Simply not allowing a definition where there are two tags that are not mutually exclusive is too restrictive. The definition above is exactly the one we want. One possibility is for the order in which the tags are defined to be important. A tag definition then means, match against this tag only when those above have failed to match the candidate value. A tag that appears below one with which it is not mutually exclusive must have an argument type that is less general than the argument type of the earlier tag. In practice the earlier tag usually has no argument and hence any type is less general than it. Where two tags are mutually exclusive the order of the matches against the values is not enforced.

Currently a tagged value is given the type corresponding to the type within which the constructors that may have been used to create the value were defined. We would like instead for a tagged value to be given a type which represents the set of constructors that were possibly used to create the value. This situation is similar to the use of polymorphic variants within OCaml [57]. The named types in which these constructors are defined can then be convenient synonyms for the case where the set of constructors contains all the constructors for that type. Hence we obtain typings such as:

```
let v = Null
 v : {Null}

let v2 = if ... then Null else Long 1
 v2 : { Null | Long }

let is_null v =
   match v with
     Null -> true
   | Ptr _ -> false
is_null : { Null | Ptr } -> bool

let b2 = is_null v
b2 : bool

(*
   v2 may have been created using the Long constructor but
   the function is_null does not test against this
*)
let b1 = is_null v2
```

```
type_error : ...
```

The final definition here raises a type error, because we attempt to apply a function to a value which may have been constructed with a constructor that is not matched against by the function.

It now becomes clear that our overall type system would be greatly enhanced by allowing subtyping. We can see the type `{Ptr | Long}` as a subtype of `{Ptr | Long | Null }` and similarly `{Ptr | Long | Null} -> bool` as a subtype of `{Ptr | Long} -> bool`. Hence our typing of masked constructor tags would benefit from being used in a subtyping type system since functions that do not first check for `Null` can be written, but only applied to arguments that could not have been created with `Null`. The next chapter explores this in more detail.

Some types simply do not make sense and should be rejected. The example shown here is useful due to the fact that the `Null` constructor takes no arguments, hence it is safe to match against this constructor when the argument is possibly a `heap_pointer`. In this case a `heap_pointer` would probably never equal 0, however even if it somehow did, we would not lose any safety by falsely matching the `Null` tag. To see this, imagine that we defined the `heap_pointer` type to be **`int`**. We could then construct a value as `Ptr 0`, this would match against the `Null` even though it was created with the `Ptr` constructor. However there would be no harm, at least in terms of type safety, to do so. An example of an illogical type that should be rejected - by applying the rule that a tag which is not mutually exclusive with an earlier tag must have a subtype for its argument type - would be the following:

```
type illogical =
    String {0 with mask 1} of string
  | Int {0 with mask 1} of int
```

It should be noted that the task of the type checker to check whether or not two tags are mutually exclusive is not always trivial; consider the following definition:

```
type immediate silly =
    I { 2 with mask 2 } of int
  | S { 0 with mask 1 } of string
```

Any value with the two least significant bits set to `10` will match both tags.

Finally we make an addition to the syntax to allow a tag that need not be matched. We use the underscore to denote this. It is equivalent to a tag of `{ 0 with mask 0}`, in that it will match any value. It is useful when there are some special values of a type that should be matched against while anything else can be considered as belonging to some other type. A good example is possibly null C-style pointers, for example a C string.

```
type immediate c_string =
    Null_pointer { 0 }
  | C_string { _ } of char_string
```

Where we have defined `char_string` to represent C strings that cannot be null. Underscore tags do not change our requirements of the type system, a tag that uses the underscore is subject to the same restraints that one using a tag of `{ 0 with mask 0}` would be, hence generally an underscore tag is given within the last constructor of a type, and is matched against last as all other constructors must be checked first.

4.2.6 Tag operations

In the section above, the `Long` constructor was introduced. The constructor though, matched against the format of a `Long` value and left it as it was. This was not quite what we wanted, because to get the true value of the integer value represented by the `Long` value we had to remember to shift the value one place to the right. Similarly when creating a value with the `Long` constructor we have to remember to shift the value one place to the left and add 1. This resulted in the line for printing OCaml values looking like this:

```
    | Long i -> print_int (i >> 1)
```

Clearly the definition of the constructor should be able to convey this information so that these operations can be performed automatically by the pattern match. We also want the inverse of these operations to be performed when we create the value. To enable this, Nitro provides tag operations. These are affixed within a constructor definition, after the description of a tag and before the argument type. Tag operations are a restricted set of expressions with an associated inverse operation. The type of an OCaml value can now be given as:

```
type immediate ocaml_value =
    Ptr   { 0 with mask 1 } of heap_object
  | Long { 1 with mask 1 } (>>. 1) of int
```

The `(>>. 1)` gives the tag operation that must be performed on the argument of the `Long` tag after we have matched the tag but before we match the argument against any pattern. The associated inverse operation is applied to arguments of a constructor when creating a value. This is done before we test that the value is a valid argument to the constructor. The dot after the shift operation in the above example tells the compiler that the inverse operation is to shift left and pad out with ones rather than zeros.

4.2.6.1 Typing

The tag operations introduce no extra typing at compile time. However, extra dynamic checks are inserted at run time to ensure that the value does not lose information when the creation tag operation is used and its inverse is used to deconstruct the value. For example when shifting such that the inverse shift operation pads with zeros, the compiler must insert a check that the argument value to which the constructor is being applied does indeed have zeros in those bits which are shifted out. This ensures that when the inverse operation is applied upon using the constructor to deconstruct the value, the argument obtained is the same value as that which was applied, and hence can be seen at the same type.

4.2.7 Multiple arguments packed in a tag

As mentioned above, it is often the case that the number of different tags is rather small compared with the amount possible from the word size of the machine. In addition the arguments may not require a whole word. When this situation arises it is tempting to store the tag of the value and its argument in the same word. We actually had something similar to this when we used the least significant bit of an `ocaml_value` to flag whether the value was a pointer or not. To store multiple tags is always possible by using intermediate types and shifting. Sometimes though this is inconvenient when we want to access only one of the values stored within a value without going through several layers of constructors. One example is the header value for an OCaml heap object. Heap objects in OCaml are stored with a single header value which describes the kind of the object(s) following the header. The header value contains three pieces of information. Firstly it stores an OCaml tag. An OCaml tag is one of several defaults that indicate the following bytes represent a string, a function closure, a record value or one of the other OCaml base types. In the case that the value is a tagged union type, then the tag is the numerical representation of the constructor used to create the value. The second piece of information stored is a further tag used by the garbage collector to say what state the value is in. This is known as the *colour* of the value. Finally the header value stores how large the heap allocated object is.

To represent a heap object in Nitro then, we use multiple arguments within a tag, since this allows us to access these three pieces of information in any order. Here is the definition:

```
type heap_object =
    Closure {247 with mask 0xFF;
                _ (mask 0x300 >> 8) of gc_colour;
                _ (>> 10) of int; }
  | String { 252 with mask 0xFF;
                 _ (mask 0x300 >> 8) of gc_colour;
                 _ ( >> 10) of int}

  | Double { 253 with mask 0xFF;
                _ (mask 0x300 >> 8) of gc_colour;
             } precedes (int * int)

and immediate ocaml_value =
    Ptr { 0 with mask 1 } of heap_object
```

```
| Long { 1 with mask 1 } (>>. 1) of int
```

Note that the arguments within the braces of the tag only refer to the portion of the tag masked by the associated argument. This is in contrast to those arguments outside the braces that refer to the entire tag value. Generally the arguments within the braces are of type integer or a related type since pointers cannot in general be stored in less than a word. In any case the compiler inserts a check that the arguments given to a constructor when used to create a value do not lose information when masked upon (or other tag operation) to fit the values into the argument. So for example we could use this to utilise the least significant bits in a pointer type where the pointers are always word aligned - whether this reads better than having the pointer argument outside the braces and hence referring to the whole tag value is a matter of personal taste.

4.2.7.1 Typing

A valid type definition containing multiple arguments in Nitro does not contain multiple arguments within a single constructor which overlap. The constructors must still follow the rules for overlapping constructors in that any two which are not mutually exclusive must either have the same argument type, or the constructor with the more general argument type is matched against first in any pattern matching. The wild card argument tags have no effect on whether or not a constructor overlaps with another since that portion of the tag may be any value.

4.2.8 Bare Arrays

This section describes how arrays are handled in Nitro. The chosen representation must allow foreign arrays to be accessed, but there must also be an array format to native to Nitro which is convenient to use. The array format described here allows both such without requiring two separate array representations. Therefore foreign and native arrays need not be distinguished from each other, for example, by storing them separately.

4.2.8.1 Motivation

Arrays are a crucial part of the foreign data interface. If one is to access foreign values, one must be able to access an array of foreign values. A type-safe language however must ensure that an access to an array is made within the bounds of the array.

In a high-level language safe array access is commonly achieved by storing the length of the array together with the array. Any index into the array is checked (at runtime) to be within the bounds of the array by comparing the index value with zero and the length stored with the array. In the case that the index is outside the bounds of the array then either an exception or a fatal error is raised.

This arrangement is not an option for external arrays, since while the length may be stored with the array, it may not be in the format expected. Certainly there exist several language implementations which store the length of the array in different ways, hence it is not possible to access all kinds of arrays with that technique.

A more general method of representing arrays is to allow the user to describe to the compiler how the length is stored. The user may then access the elements of arrays of different representations and the compiler can ensure that such accesses are safe.

This scheme is a compromise between two styles of array access. The unsafe version in which the user must do all of the array bounds checking themselves and the abstracted version in which the storing of the array length and the bounds checks are automatically inserted by the compiler. In Nitro the checking of array indexing operations is done explicitly by the programmer but the compiler checks that the programmer has done it correctly. Note that the compromise is in the flexibility not in any ability to avoid array bounds checks because the programmer believes that the index will never be invalid. However it could be argued that the extra information available to the compiler may allow an optimiser to remove more unnecessary bounds checking. This possiblity is not explored in the current text.

4.2.8.2 Details

To realise this scheme in Nitro there is the concept of a *bare array*. The main idea is that the user must provide the length of the array in order to make an array access. The type of a length expression is a new type, called an *index type*, which the compiler can treat separately to ensure that the correct length expression is given for each array access.

An access to a bare array is written as `a.(e).[i]`, where `a` is the array to be accessed, `e` is the length of the array and `i` is the index into the array. In the common case the expression `e` will be an index variable, but in general could be any expression with index type.

The type system must ensure that `e` is indeed a valid representation of the length of `a`. To achieve this the type of a bare array has an associated index variable. Index variables act as existential type variables; they cannot match any other index variable but themselves.

An index type on its own is written **`index`**`.(v)` and as part of an array type as τ **`array`**`.(v)`, where τ is the type of the elements within the array.

To create an array we use the **`array`** e_1 e_2 expression, where e_2 is the length of the new array and must have index type. The index variable of the index type is then associated with the resulting array type. The expression e_1 is evaluated to give the value with which all the elements of the array are initialised.

When an access to a bare array is made, such as e_1`.(len).[`e_2`]`, the type system forces the expression e_1 to have the type of a bare array, and `len` must have an index type. Furthermore the index variable within that index type must be the same as the index variable associated with the array type. The expression e_2 must have type **`int`**. No check is made at compile time regarding its size and hence suitability to index into the array e_1. At runtime the value is checked against `len` which is known to hold the length of the array.

Values of type index, used to create arrays and as the tests, are created with the **`let index`** construct which has the form **`let index`** `v =` e_1**`in`** e_2. The expression e_1 must have type integer. The name `v` is added to the current typing environment with an index type containing a new index variable. The

```
let make element size =
  let index s = size in
    (s, array element s)

: 'a -> int -> (index.(s), 'a array.(s))

let access a i =
  let (s, data) = a in
    data.(s).[i]

: (index.(s), 'a array.(s)) -> int -> 'a

let length a  =
  let (s, data) = a in
    #s

: (index.(s), 'a) -> int
```

Figure 4.2: Code for safe arrays as pairs in Nitro.

expression e_2 is typed with this new typing environment to give the type of the whole expression.

We can convert from indexes into integers with the # operator. In the expression $\#e_1$, the expression e_1 must have an index type and the whole expression is given type **int**. Note that both operations can be compiled out at compile time and hence incur no runtime cost. The operations are necessary only to give enough information to the type checker.

4.2.8.3 Safe Arrays

As an example *safe arrays* – that is arrays which are stored together with their length for bounds checking – may be implemented in Nitro using bare arrays in a pair. The first value is the length of the bare array which is the second value. Functions can then be written to create, access and take the length of this pair representation of arrays. The Nitro code for this is given in Figure 4.2.

The final function `length` has a very general type inferred for it that gives no information about the second element of the pair and does not enforce that it is called with an array at all. One could give the function a type constraint such that the type is `(index.(s), 'a array.(s)) -> int.`

The following section highlights two problems with the pair representation of arrays and details a solution.

4.2.8.4 Marked Arrays

Requiring the programmer to provide an index variable each time an array access is made is an extra burden on the programmer. In the last example we saw that we can build on the bare arrays to create a more high-level representation. This is called a *marked array*. The intention is that the programmer may use marked arrays in the course of normal programming and need only resort to the bare array representation when required for accessing foreign arrays.

A marked array format then becomes the native array kind in Nitro, which avoids two distinct representations of arrays; one for use with foreign arrays and one for internal Nitro programming. Instead the native array format is built upon the foreign array format.

Bare arrays can also be used to optimise other representations, for example matrices can store the length of all of the arrays only once rather than once for each row.

The above representation of safe arrays used a pair to store the length together with the bare array. There are two disadvantages with this representation of marked arrays. Firstly as noted above the type inferred for the `length` function is more general than desired.

The second disadvantage is the restriction that the types of two high-level arrays with different lengths represented in this way cannot be unified. This is because index variables cannot be unified with anything other than themselves. This means that the pair representation of marked arrays is insufficient for use as the native kind of arrays in Nitro.

Both problems can be solved by defining a high-level array using a tagged type with a single constructor. As in:

```
type 'a marked_array s = Array of (index.(s), 'a array.(s))
```

Because the index variable s is existential such an array representation can be seen as an abstract type in the style of [58]. Applying the constructor Array gives a value of type 'a marked_array which hides the index variable used to create the array. Hence two marked_array types can be unified together. Matching against the Array constructor brings a new index variable into scope. The new index variable is attached to both the index type of the length and the bare array type. Hence safe array accesses can still be made, such as:

```
let access a i =
  let Array (s, data) = a in
    data.(s).[i]

: 'a marked_array -> int -> 'a
```

The length function can also be rewritten as:

```
let length a =
  let Array (s, _) = a in #s

: 'a marked_array -> int
```

Notice that the use of index variables is entirely hidden by the type of the access function. The revised access function has the required type, namely that it may only be applied to marked arrays. The inferred type for the re-written length function also rejects any attempt to apply it to something other than a marked array.

In a similar manner functions may be written for the common array operations such as iter and copy. These can be grouped together to form a module with the actual type of the marked array hidden. The programmer can then use these arrays as conveniently as arrays are used in a traditional high-level language, without the need to provide a check variable at each index operation.

```
type gc_info
type immediate 'a m_array len =
  Array { _ } of (int.(len), gc_info, 'a array.(len))

let index_marray a i =
  let Array (l, _, data) = a in
      data.(l).[i]

: 'a m_array -> int -> 'a
```

Figure 4.3: The definitions for accessing M arrays.

4.2.8.5 Foreign Arrays

The preceding section has shown that convenient marked arrays can be defined in terms of the bare arrays which Nitro provides. This chapter though is concerned with the ability to describe the representation of external arrays in order to access them.

Suppose there is an array representation in some foreign language M. Arrays in M consist of a tuple of three elements, the first is the length, the second is some information used by the garbage collector, and the third is the actual array itself.

The definitions given for marked arrays in the preceding section wrap the arrays within a constructor. Recall that this was done to allow the existential types involved with the arrays to be hidden and hence arrays of different lengths considered to have the same type. A foreign array however cannot be wrapped in a constructor since this will change the representation. To avoid this problem an **immediate** constructor is used. The definitions for the type of M arrays are given in Figure 4.3

Now suppose there is a further foreign language N also exporting arrays which must be accessed. These arrays are similar but contain a further tuple element before the actual array, which contains debugging information. Such arrays can be defined as in Figure 4.4.

```
type gc_info
type debug_info
type immediate 'a n_array =
   Array { _ } of (int.(len), gc_info, debug_info, 'a array.(len))

let index_narray a i =
    let Array (l, _, _, data) = a in
      data.(l).[i]

: 'a n_array -> int -> 'a
```

Figure 4.4: The definitions for accessing N arrays.

This section has given a high-level introduction the foreign data facilities provided by Nitro. The following sections will provide a formal definition of the syntax and semantics. Section 4.6 details two example uses of the foreign data facitilies defined in this chapter and Section 4.7 ends the chapter by concluding on the approach taken to foreign data representation access.

4.3 Syntax

The syntax for the new type definitions added to Nitro to allow the definition of types representing foreign data types is given in Figure 4.5. Figure 4.6 shows the additions to the syntax of expressions, patterns and types. These syntax additions include the additional syntax necessary for bare array manipulation. Finally in addition to the definitions given here a function may be exported as an external C function using the **export** keyword.

4.3.1 Derived Forms

There are a few simple derived forms in the syntax. These are convenient for the programmer while still allowing the semantic rules to be reduced by not considering these forms separately. Recall from the previous chapter in Section

$$
\begin{array}{lcl}
\textit{tydec} & := & \textbf{type}\ \textit{tybind} \\
\textit{tybind} & := & \langle\textbf{immediate}\rangle\ \textit{tyvars}\ \textit{tyname}\ \textit{tyivars}\ =\ \textit{constrs}\langle\textbf{and}\ \textit{tybind}\rangle \\
\textit{tyvars} & := & \\
 & \mid & 'a \\
 & \mid & ('a\{;'b\}^{+}) \\
\textit{tyivars} & := & \textit{indexvar}^{*} \\
\textit{indexvar} & := & \textit{tyname} \\
\textit{constrs} & := & \textit{constr}\ \langle\mid\ \textit{constrs}\rangle \\
\textit{constr} & := & \textit{Con}\ \{\textit{tagdesc}\}\ \langle\textit{tagop}\ \textit{typearg}\rangle \\
\textit{tagdesc} & := & \textit{tagarg}^{+} \\
\textit{tagarg} & := & \textit{tagvalue}\ \textbf{with mask}\ \textit{num}\ \langle\langle\textit{tagop}\rangle\ \textbf{of}\ \textit{ty}\rangle\ ; \\
\textit{tagvalue} & := & \textit{num} \\
 & \mid & -\textit{num} \\
 & \mid & \_ \\
\textit{tagop} & := & \textit{tagoper}\ \textit{tagoperand} \\
 & \mid & \textit{tagoper}\ \textit{tagoperand}\ \textit{tagop} \\
\textit{tagoper} & := & >> \\
 & \mid & << \\
 & \mid & >>. \\
 & \mid & <<. \\
 & \mid & + \\
 & \mid & - \\
 & \mid & \mid\mid \\
 & \mid & \&\& \\
 & \mid & \textit{id} \\
\textit{tagoperand} & := & \textit{num} \\
\textit{typearg} & := & \textit{typlace}\ \textit{ty} \\
\textit{typlace} & := & \textbf{of} \\
 & \mid & \textbf{precedes}
\end{array}
$$

Figure 4.5: The syntax for foreign type definitions

$$
\begin{array}{lcl}
expr & := & \textbf{let index } i\ = expr \textbf{ in } expr \\
 & | & \#i \\
 & | & \textbf{array } expr_1\ expr_2 \\
 & | & expr_1.(expr_2).[expr_3] \\
 & | & expr_1.(expr_2).[expr_3] \leftarrow expr_4 \\
 & | & Con\ \{targexps\}\ \langle expr \rangle \\
targexps & := & expr; \langle targexps \rangle \\
pattern & := & Con\ \{targpats\}\ \langle pattern \rangle \\
targpats & := & pattern; \langle targpats \rangle \\
\tau & := & \textbf{index}.(i) \\
 & | & \tau\ \textbf{array}.(i)
\end{array}
$$

Figure 4.6: Extensions to the syntax of Nitro expressions, patterns and types.

3.1.2 that:

$phrase_1 \Longrightarrow$

$\qquad phrase_2$

Is written to mean that $phrase_1$ may be written as an abbreviation for the expanded form $phrase_2$.

In the given syntax it is not possible to omit the mask of a custom constructor. As a convenience the programmer may omit this and the full mask is inferred instead.

Con **of** $\{1\} \Longrightarrow$

$\qquad Con$ **of** $\{1$ **with mask** $0xFFF...\}$

Here the length of the mask is dependent on the size of the word of the machine.

Similarly one must always give a *tagop* if one is to provide a tag operation. However Nitro allows the identity tag operation and a constructor definition which contains an argument without a tag operation is inferred to have the identity operation. The constructor definition

Con **of** $ty \Longrightarrow$

$\qquad Con$ **of** $id\ ty$

The following derived form allows the user to leave out a tag description. Note that usually the *tagop* here will be *id* and will be the result of an expansion of the previous derived form. The *tagdesc* is arbitrary but in practice the compiler will choose a simple constant unused by any other constructor within the same tagged type definition.

Con **of** $tagop\ ty \Longrightarrow$

$\qquad Con$ **of** $\{tagdesc\}\ tagop\ ty$

When this rule is applied, all constructor application expressions and patterns concerning the constructor in question must be modified by the rules:

$Con\ e \Longrightarrow$

$\qquad Con\ \{tagdesc\}\ e$

$Con\ p \Longrightarrow$

$\qquad Con\ \{\_\}\ p$

4.4 Static Semantics

The static semantics of the core Nitro language may now be updated to include the facilities for defining the runtime representation of values. Since many of the changes are within the type definition language, the new type definition rules are presented first. These are followed by the rules for typing expressions and patterns which must be updated. Before the type declaration rules are given the new typing context and in particular the constructor environment portion is described.

4.4.1 Typing Contexts

In section 3.2.1 the structure of a type context was described including the part related to tagged union data types called the constructor environment. The information held for each constructor identifier in the constructor environment must be revised to allow for the correct typing of the Nitro foreign data interface constructs.

The data which must be held for each constructor is now:

- The parent tagged union type to which it belongs, which contains information detailing whether this type is **immediate** or not.
- The structure of the tag value itself. This structure information contains the details of each of the parts that make up the tag value (or tag arguments). For each tag argument the environment must store
  - The tag argument value
  - the tag argument mask
  - the tag argument operation
  - the type of the tag part
- The main tag operation
- The main tag type and argument placement which details whether this is a **precedes** argument or not.

Note that the tag operation and the argument placement are not present in the case that the constructor accepts no arguments outside of the tag.

4.4.2 Tagged Union Type Declarations

$\boxed{C \vdash tydec \Rightarrow C_1}$

$$\frac{C_1 = C + Tenv \qquad C_1 \vdash tybind \Rightarrow (Tenv, Fenv, Cenv)}{C \vdash \textbf{type}\ tybind \Rightarrow C_1 + Fenv + Cenv} \tag{64}$$

$\boxed{C \vdash tybind \Rightarrow (Tenv, Fenv, Cenv)}$

$$\frac{\begin{array}{c}\tau = tyname(tyvars) \qquad C, tyivars, \tau \vdash constrs \Rightarrow Cenv_1 \qquad Tenv_1 = \{tyname \mapsto tyvars\} \\ \langle C \vdash tybind \Rightarrow (Tenv_2, Fenv, Cenv_2)\rangle\end{array}}{\begin{array}{c}C \vdash \langle\langle \textbf{immediate}\rangle\rangle tyvars\ tyname\ tyivars\ =\ constrs\ \langle \textbf{and}\ tybind\rangle \\ \Rightarrow (Tenv_1\langle \cup Tenv_2\rangle, \{\}\langle \cup Fenv\rangle, Cenv_1\langle \cup Cenv_2\rangle)\end{array}} \tag{65}$$

$\boxed{C, tyivars, \tau \vdash constrs \Rightarrow Cenv_1}$

$$\frac{C, tyivars, \tau \vdash constr \Rightarrow Cenv_1 \qquad \langle C, tyivars, \tau \vdash constrs \Rightarrow Cenv_2 \rangle}{C, tyivars, \tau \vdash constr\ \langle \mid constrs \rangle \Rightarrow Cenv_1 \langle + Cenv_2 \rangle} \tag{66}$$

$\boxed{C, tyivars, \tau \vdash constr \Rightarrow Cenv}$

$$\frac{C \vdash ty \Rightarrow \tau_1 \qquad C \vdash tagargs \Rightarrow td \qquad Cenv_1 = \{Con \mapsto (td, tagop, typlace, (\tau_1 \rightarrow \tau))\}}{C, tyivars, \tau \vdash Con\ \{tagargs\}\ tagop\ typlace\ ty \Rightarrow Cenv_1} \tag{67}$$

$$\frac{C \vdash tagargs \Rightarrow td \qquad Cenv_1 = \{Con \mapsto (td, \tau)\}}{C, tyivars, \tau \vdash Con\ \{tagargs\} \Rightarrow Cenv_1} \tag{68}$$

$\boxed{C \vdash tagdesc \Rightarrow td}$

$$\frac{C \vdash tagarg \Rightarrow ta \qquad \langle C \vdash tagargs \Rightarrow tas \rangle}{C \vdash tagarg \langle tagargs \rangle \Rightarrow (ta \langle, tas \rangle)} \tag{69}$$

$\boxed{C \vdash tagarg \Rightarrow ta}$

$$\frac{C \vdash ty \Rightarrow \tau \qquad ta = (tagvalue, num, tagop, \tau)}{C \vdash tagvalue\ \textbf{with mask}\ num\ tagop\ \textbf{of}\ ty\ ; \Rightarrow ta} \tag{70}$$

4.4.2.1 Additional Restrictions

In rule 67 the compiler rejects the constructor definition if $typlace =$ **precedes** and τ_1 is a ground type.

In rule 69 the tag value portion of each of the tag arguments must not overlap. This can be checked by performing a bit-wise *and* operation between any two of the tag argument mask values.

4.4.3 Constructor Application

The typing context stores the information required of the applied constructor. The information is of the following form: $(tagargs\langle, tagop, typlace\rangle, \tau)$. The *tagop* and *typlace* components are present if the constructor accepts an argument and absent otherwise. The *tagargs* component is a list of components which are themselves made up of several sub-components. A single *tagarg* has the form: $(tagvalue, num, tagop, \tau)$. In the following rules the phrase "*types* **in** *tagargs*" will mean the list of τ components got from extracting the τ component from each of a list of *tagargs*.

$\boxed{C \vdash e \Rightarrow \tau}$

$$\frac{\begin{array}{c}\mathcal{T}(C, Con) = (tagargs, tagop, typlace, (\tau_1 \rightarrow \tau)) \\ C \vdash e \Rightarrow \tau_1 \qquad \tau s = types \text{ in } tagargs \qquad C \vdash targexps \Rightarrow \tau s\end{array}}{C \vdash Con\ \{targexps\}\ e \Rightarrow \tau} \tag{71}$$

$$\frac{\begin{array}{c}\mathcal{T}(C, Con) = (tagargs, \tau) \\ \tau s = types \text{ in } tagargs \qquad C \vdash targexps \Rightarrow \tau s\end{array}}{C \vdash Con\ \{targexps\}\ \Rightarrow \tau} \tag{72}$$

$\boxed{C \vdash targexps \Rightarrow \tau s}$

$$\frac{C \vdash targexp \Rightarrow \tau \qquad \langle C \vdash targexps \Rightarrow \tau s \rangle}{C \vdash targexp; \langle targexps \rangle \Rightarrow \tau; \langle \tau s \rangle} \tag{73}$$

4.4.4 Pattern Matching

$\boxed{C \vdash pattern\ \Rightarrow (Venv, \tau)}$

$$\frac{\begin{array}{c}C \vdash pattern\ \Rightarrow (Venv_1, \tau_1) \qquad \mathcal{T}(C, Con) = (tagarg^{(k)}, tagop, typlace, (\tau_1\ \rightarrow\ \tau)) \\ C, tagarg^{(k)} \vdash targpats^{(k)} \Rightarrow Venv_2\end{array}}{C \vdash Con\ \{targpats^{(k)}\}\ pattern\ \Rightarrow (Venv_1 + Venv_2, \tau)} \tag{74}$$

$$\frac{\mathcal{T}(C, Con) = (tagarg^{(k)}, \tau) \qquad C, tagarg^{(k)} \vdash targpats^{(k)} \Rightarrow Venv_2}{C \vdash Con\ \{targpats^{(k)}\}\ \Rightarrow (Venv_1 + Venv_2, \tau)} \tag{75}$$

$\boxed{C, tagarg^{(k)} \vdash targpat^{(k)} \Rightarrow Venv}$

$$\frac{C \vdash targpat \;\Rightarrow (Venv_1, \tau) \qquad \tau = type\ of\ tagarg \qquad \langle C, tagarg^{(k)} \vdash targpat^{(k)} \Rightarrow Venv_2 \rangle}{C, tagarg; \langle tagarg^{(k)} \rangle \vdash targpat; \langle targpat^{(k)} \rangle \Rightarrow Venv_1 \langle + \ Venv_2 \rangle} \tag{76}$$

4.4.5 Additional Constraints on Matches

The rules for match expressions remain the same. The compiler will reject all those match expressions for which the rules do not permit a type to be inferred. However there is another class of match expressions which must be rejected by the compiler. Those involvling ambiguous matches which are unsafe due to overlapping tag values.

In section 4.2.5.2 it was explained that it is often useful to allow two custom constructors to overlap. Whereby overlapping constructors are two constructors for which at least one value may match either constructor. It was noted that this is useful but also unsafe to allow without restriction. The null pointer type is such an example:

```
type immediate poss_null_c_string =
    Null { 0 }
  | Ptr  { _ } of cstring
```

Here the `Ptr` constructor may match any value. The compiler must reject any pattern match where a `Ptr` pattern may be matched against a value which has not (in the same match expression) previously been tested so as to ensure that the value does not match the `Null` constructor.

This means that the sets of match rules depicted in Figure 4.7 are rejected.

Notice that in the last set the match rules are rejected even though the `Null` value is matched against. This is because the enclosing pattern may fail despite the `Null` pattern succeeding.

4.4.6 Bare Arrays

$$\frac{C \vdash e_1 \Rightarrow \textbf{int} \qquad C_i[i \mapsto \textbf{index}.(a)] \vdash e_2 \Rightarrow \tau \qquad a \notin C}{C \vdash \textbf{let index}\ i\ = e_1\ \textbf{in}\ e_2 \Rightarrow \tau} \tag{77}$$

```
let Ptr p = e1 in e2

match e of
    Ptr p -> e1
end

match e of
    Ptr p -> e1
  | Null   -> e2
end

match e of
    (Null, 1) -> e1
  | (Ptr, x)   -> e2
end
```

Figure 4.7: Matches rejected because *Ptr* may match a *Null* value.

$$\frac{C \vdash i \Rightarrow \textbf{index}.(i)}{C \vdash \#i \Rightarrow \textbf{int}} \tag{78}$$

$$\frac{C \vdash e_1 \Rightarrow \tau \qquad C \vdash e_2 \Rightarrow \textbf{index}.(i)}{C \vdash \textbf{array}\ e_1\ e_2 \Rightarrow \tau\ \textbf{array}.(i)} \tag{79}$$

$$\frac{C \vdash e_1 \Rightarrow \tau\ \textbf{array}.(i) \qquad C \vdash e_2 \Rightarrow \textbf{index}.(i) \qquad C \vdash e_3 \Rightarrow \textbf{int}}{C \vdash e_1.(e_2).[e_3] \Rightarrow \tau} \tag{80}$$

$$\frac{C \vdash e_1 \Rightarrow \tau\ \textbf{array}.(i) \qquad C \vdash e_2 \Rightarrow \textbf{index}.(i) \qquad C \vdash e_3 \Rightarrow \textbf{int} \qquad C \vdash e_4 \Rightarrow \tau}{C \vdash e_1.(e_2).[e_3] \leftarrow e_4 \Rightarrow ()} \tag{81}$$

4.5 Dynamic Semantics

In this section the dynamic semantics are updated to include the runtime representation control facilities of Nitro. The main job of the compiler is to produce executable code which will ensure runtime properties which cannot be ensured at compile time. In particular when a value is constructed using a tagged union datatype constructor, it must be possible to retrieve the same value during a pattern match deconstruction. If this were not the case then the type system would be unsound, since a value constructed at one type could be deconstructed to a value of another type.

Since there are many methods which may be combined to control the way a value is represented, it is possible that many such runtime checks must be inserted. However in practice only one such method or a simple combination of methods is used and hence the compiler need only insert one or, in many cases, no checks.

Before continuing with the inference rules for the dynamic semantics the reader is reminded of the state and *STOP* conventions used in the core Nitro dynamic semantics and first defined in section 3.3.4.

4.5.1 The Fail Convention

The pattern matching rules, given in section 4.5.5, are all subject to a *FAIL* convention which is analogous to the *STOP* convention. Each premise which may produce a value environment or a *FAIL* causes the main rule to be duplicated with both the premise and the conclusion producing a *FAIL*. Therefore the *FAIL* convention causes the top rule 93 to produce two further rules:

$$\frac{E(Con) = (tagargs, op, typlace) \qquad E, v_1 \vdash tagargs, targpats \Rightarrow FAIL}{E, [(v_1, v_2)] \vdash Con\ \{targpats\}\ pattern \Rightarrow FAIL} \tag{82}$$

$$\frac{E(Con) = (tagargs, op, typlace) \qquad E, v_2 \vdash pattern \Rightarrow FAIL}{E, [(v_1, v_2)] \vdash Con\ \{targpats\}\ pattern \Rightarrow FAIL} \tag{83}$$

4.5.2 Values

Values are the same as in the core Nitro dynamic semantics given in Section 3.3 with the following addition.

A group of values may be 'boxed' using $[v_1, \ldots, v_n]$ meaning that each of the values are stored in consecutive memory locations.

4.5.3 Evaluation Contexts

An evaluation context must store dynamic information about the program. This means that the bound variables within a program will be mapped by the evaluation context to their respective values. Additionally an evaluation context stores information about the tagged union datatype constructors. The information returned for a given constructor is of the form:

$$(\langle\mathbf{immediate},\rangle tagargs\langle\langle, op, typlace\rangle\rangle)$$

The presence of the **immediate** flag specifies that the constructor is an immediate one. Where there is an argument to the constructor (other than the tag arguments), the *op* and *typlace* components are present.

In addition there is the state or memory of a program. The memory of the program is a mapping from addresses to values. Due to the state convention the state is only explicitly mentioned within rules which must modify or examine the memory of the program.

Function application syntax will be used to interrogate the evaluation context so that where E is an evaluation context and Con is a constructor identifier then $E(Con)$ will return the information stored about the tagged union constructor.

4.5.4 Constructor Application

Constructor application must use the information in the evaluation context about the given constructor to create the correct representation of the value. In addition the argument values must be checked to ensure that they are suitable for encapsulation by the given constructor. Such checks are those which cannot be performed ahead of time by the compiler in accordance with the static semantic rules. The semantics given here are conservative in that such a check is the most general check and is always performed. A compiler however would not need to be very sophisticated to reduce the check to a simpler one or remove it completely.

The dynamic semantics only consider programs which are given a type by the rules for the static semantics. Therefore a situation such as a constructor application without an argument given but where the constructor in the evaluation context specifies an argument operation and placement cannot occur and are not considered by the following rules.

4.5.4.1 Tag operations and Checks

There are two operations, other than the tag operators themselves, used by the dynamic semantics for constructor application. The $||$ and the & operations. The first combines two values together using a bitwise 'or' operation and the second combines two values using a bitwise 'and' operation. The first is used for immediate arguments to combine the immediate argument with the constructor (which may contain tag arguments). The second is used to apply the masks to the tag arguments.

When a constructor is applied, a runtime check is performed to ensure that the construction of the tagged value has not lost information of the argument. If this were allowed then it may be possible to construct a value using an argument of the correct type which, when that argument is extracted from the tagged value, is distorted and hence may not be of the appropriate type. For example in the following code:

```
let Con x = Con v in ...
```

It must be the case that x is equal to v otherwise there is no way to be sure of the type of x. The generic test which is inserted is a test that the same value is extracted as was used to construct. A compiler however may choose to optimise or even remove such checks in specific cases where it is safe to do so. The rule 86 uses the check $v_2 = op_r(v)$ to perform this dynamic runtime check.

Note that in rule 87 the restriction $v_2 = op_r(v)$ is very strong. In practice few foreign constructs require complex tag arguments with an immediate constructor which also has an argument. This means that few constructor applications are failed due to this restriction.

$\boxed{E \vdash e \Rightarrow v, STOP}$

$$\frac{E(Con) = (tagargs) \qquad E, tagargs \vdash targexps \Rightarrow v}{E \vdash Con\{targexps\} \Rightarrow [v]} \tag{84}$$

$$\frac{E(Con) = (\mathbf{immediate}, tagargs) \qquad E, tagargs \vdash targexps \Rightarrow v}{E \vdash Con\{targexps\} \Rightarrow v} \tag{85}$$

$$\frac{\begin{array}{c} E(Con) = (tagargs, op, typlace) \qquad E, tagargs \vdash targexps \Rightarrow v_1 \\ E \vdash e \Rightarrow v_2 \qquad v = op(v_2) \qquad v_2 = op_r(v) \end{array}}{E \vdash Con\{targexps\}\ e \Rightarrow [v_1, v]} \tag{86}$$

$$\frac{\begin{array}{c} E(Con) = (\mathbf{immediate}, tagargs, op, typlace) \qquad E, tagargs \vdash targexps \Rightarrow v_1 \\ E \vdash e \Rightarrow v_2 \qquad v = v_1 \| op(v_2) \qquad v_2 = op_r(v) \end{array}}{E \vdash Con\{targexps\}\ e \Rightarrow v} \tag{87}$$

$$\frac{\begin{array}{c} E(Con) = (tagargs, op, typlace) \qquad E, tagargs \vdash targexps \Rightarrow v_1 \\ E \vdash e \Rightarrow v_2 \qquad v_1 \neq op_r(op(v_2)) \end{array}}{E \vdash Con\{targexps\}\ e \Rightarrow STOP} \tag{88}$$

$$\frac{\begin{array}{c} E(Con) = (\mathbf{immediate}, tagargs, op, typlace) \qquad E, tagargs \vdash targexps \Rightarrow v_1 \\ E \vdash e \Rightarrow v_2 \qquad v_2 \neq op_r(v_1 \| op(v_2)) \end{array}}{E \vdash Con\{targexps\}\ e \Rightarrow STOP} \tag{89}$$

$\boxed{E, tagargs \vdash targexps \Rightarrow v/STOP}$

$$\frac{E, tagarg \vdash targexp \Rightarrow v_1 \qquad \langle E, tagargs \vdash targexps \Rightarrow v_2 \rangle}{E, tagarg; \langle tagargs \rangle \vdash targexp; \langle targexps \rangle \Rightarrow v_1 \langle \| v_2 \rangle} \tag{90}$$

$\boxed{E, tagarg \vdash targexp \Rightarrow v/STOP}$

$$\frac{E \vdash e \Rightarrow v_1 \qquad v = (num\ \&\ (op_r(v_1)))}{E, (\_\ \textbf{with mask}\ num\ op\ \textbf{of}\ ty \vdash e \Rightarrow v} \tag{91}$$

4.5.5 Pattern Matching

The rules in this section define the semantics of matching values against patterns involving foreign data type constructors.

$\boxed{E,v \vdash Con\ \{targpats\}\ pattern \Rightarrow VE/FAIL}$

$$\frac{E(Con) = (tagargs) \qquad E,v,tagargs \vdash targpats \Rightarrow VE}{E,[v] \vdash Con\ \{targpats\}\ \Rightarrow VE} \tag{92}$$

$$\frac{\begin{array}{c} E(Con) = (tagargs, op, typlace) \\ E,v_1,tagargs \vdash targpats \Rightarrow VE_1 \qquad E,op_r(v_2) \vdash pattern \Rightarrow VE_2 \end{array}}{E,[(v_1,v_2)] \vdash Con\ \{targpats\}\ pattern \Rightarrow VE_1 + VE_2} \tag{93}$$

$$\frac{E(Con) = (\textbf{immediate}, tagargs) \qquad E,v,tagargs \vdash targpats \Rightarrow VE}{E,v \vdash Con\ \{targpats\}\ \Rightarrow VE} \tag{94}$$

$$\frac{\begin{array}{c} E(Con) = (\textbf{immediate}, tagargs, op, typlace) \\ E,v,tagargs \vdash targpats \Rightarrow VE_1 \qquad E,op_r(v_2) \vdash pattern \Rightarrow VE_2 \end{array}}{E,v \vdash Con\ \{targpats\}\ pattern \Rightarrow VE_1 + VE_2} \tag{95}$$

$\boxed{E,v,tagargs \vdash targpats \Rightarrow VE/FAIL}$

$$\frac{E,v,tagarg \vdash targpat \Rightarrow VE_1 \qquad \langle E,v,tagargs \vdash targpats \Rightarrow VE_2\rangle}{E,v,(tagarg;\langle tagargs\rangle) \vdash (targpat;\langle targpats\rangle) \Rightarrow VE_1\langle +VE_2\rangle} \tag{96}$$

$\boxed{E,v,tagarg \vdash targpat \Rightarrow VE/FAIL}$

$$\frac{v_1 = tagop_r(v\ \&\ num) \qquad E,v_1 \vdash pattern \Rightarrow VE}{E,v,(\_\ \textbf{with mask}\ num\ tagop\ \textbf{of}\ ty) \vdash pattern \Rightarrow VE} \tag{97}$$

4.5.6 Tagged Type Declarations

In the core Nitro dynamic semantics there was no need to give a semantics to type definitions. The type definitions were used only in the static semantics. However in the foreign dynamic semantics, because the type declarations affect how tagged union type values are created, information from them must be collected. Here are the dynamic semantics for the tagged union data type declarations, the purpose of these rules is to ensure that the evaluation context has enough information about the constructors used in a program.

Rule 100 uses the function *imm* which maps all constructor entries in a given constructor environment to an immediate version of the same constructor. This function implements the mappings:

$(tagargs) \longrightarrow (\textbf{immediate}, tagargs)$

and

$(tagargs, op, typlace) \longrightarrow (\textbf{immediate}, tagargs, op, typlace)$

$\boxed{E \vdash tydec \Rightarrow E}$

$$\frac{E \vdash tybind \Rightarrow CE}{E \vdash \textbf{type}\ tybind \Rightarrow E + CE} \tag{98}$$

$\boxed{E \vdash tybind \Rightarrow CE}$

$$\frac{E \vdash constrs \Rightarrow CE_1 \qquad \langle E \vdash tybind \Rightarrow CE_2 \rangle}{E \vdash tyvars\ tyname\ tyivars\ =\ constrs\ \langle \textbf{and}\ tybind \rangle \Rightarrow CE_1 \langle + CE_2 \rangle} \tag{99}$$

$$\frac{E \vdash constrs \Rightarrow CE_1 \qquad \langle E \vdash tybind \Rightarrow CE_2 \rangle \qquad CE = imm(CE_1)}{E \vdash \textbf{immediate}\ tyvars\ tyname\ tyivars\ =\ constrs\ \langle \textbf{and}\ tybind \rangle \Rightarrow CE\ \langle + CE_2 \rangle} \tag{100}$$

$\boxed{E \vdash constrs \Rightarrow CE}$

$$\frac{E \vdash constr \Rightarrow CE_1 \qquad \langle E \vdash constrs \Rightarrow CE_2 \rangle}{E \vdash constr \langle \mid constrs \rangle \Rightarrow CE_1 \langle + CE_2 \rangle} \tag{101}$$

4.5.7 Constructor Definitions

$\boxed{E \vdash constr \Rightarrow Con \mapsto Centry}$

$$\frac{}{E \vdash Con\ \{tagargs\} \Rightarrow Con \mapsto tagargs} \tag{102}$$

$$\frac{}{E \vdash Con\ \{tagargs\}\ op\ typlace\ \tau \Rightarrow Con \mapsto (tagargs, op, typlace)} \tag{103}$$

4.5.8 Bare Arrays

$\boxed{E \vdash e \Rightarrow v/STOP}$

$$\frac{E(i) = v}{E \vdash \#i \Rightarrow v} \tag{104}$$

$$\frac{E \vdash e_1 \Rightarrow v_1 \qquad E + \{i \mapsto v_1\} \vdash e_2 \Rightarrow v}{E \vdash \textbf{let index}\ i\ = e_1\ \textbf{in}\ e_2 \Rightarrow v} \tag{105}$$

$$\frac{\begin{array}{c} s, E \vdash e_1 \Rightarrow v_1, s_1 \qquad s_1, E \vdash e_2 \Rightarrow n, s_2 \\ v = [v_1, \ldots v_1] \qquad a \notin Dom(s_2) \qquad length(v) = n \end{array}}{s, E \vdash \textbf{array}\ e_1\ e_2 \Rightarrow a, s_2 + \{a \mapsto v\}} \tag{106}$$

$$\frac{\begin{array}{c} s, E \vdash e_1 \Rightarrow a, s_1 \qquad s_1, E \vdash e_2 \Rightarrow n, s_2 \qquad s_2, E \vdash e_3 \Rightarrow i, s_3 \\ 0 \leq i < n \qquad s_3(a) = [v_0, \ldots, v_{n-1}] \end{array}}{s, E \vdash e_1.(e_2).[e_3] \Rightarrow v_i, s_3} \tag{107}$$

$$\frac{\begin{array}{c} s, E \vdash e_1 \Rightarrow a, s_1 \qquad s_1, E \vdash e_2 \Rightarrow n, s_2 \qquad s_2, E \vdash e_3 \Rightarrow i, s_3 \qquad s_3, E \vdash e_4 \Rightarrow v, s_4 \\ 0 \leq i < n \qquad s_4(a) = [v_0, \ldots, v_{n-1}] \qquad v_a[i] = v \qquad v_a[j] = s_4(a)[j], j \neq i \end{array}}{s, E \vdash e_1.(e_2).[e_3] \leftarrow e_4 \Rightarrow (), s_4[a \mapsto v_a]} \tag{108}$$

$$\frac{\begin{array}{c} s, E \vdash e_1 \Rightarrow a, s_1 \qquad s_1, E \vdash e_2 \Rightarrow n, s_2 \qquad s_2, E \vdash e_3 \Rightarrow i, s_3 \\ \langle s_3, E \vdash e_4 \Rightarrow v, s_4 \rangle \qquad (i < 0)\ \text{or}\ (i \geq n) \end{array}}{s, E \vdash e_1.(e_2).[e_3]\langle \leftarrow e_4 \rangle \Rightarrow STOP} \tag{109}$$

4.6 Examples

In this section two example uses of the foreign data interface for Nitro described in this chapter are presented.

4.6.1 Ocaml

The first example develops an interface to the OCaml [11] programming language. In this section a Nitro interface to the OCaml data representation will be described. The current C interface to the OCaml data representation is also described and both are used to create an equality function for use in OCaml programs. The two approaches are then compared.

In general an interface to a language is specific to an implementation of that language. This is often offset if the language definition defines the representation of values. Alternatively there may be a standard for interfacing with, or exporting data to, the outside world. Interfacing to the OCaml language avoids this problem altogether because there exists only one implementation.

The OCaml language is a high-level functional language, which can be compiled to native code by an optimising compiler or bytecode that is interpreted by a virtual machine. The virtual machine is implemented in C and performs the interpretation of the program's bytecodes. In addition further routines, also written in C, provide runtime services such as garbage collection and polymorphic comparison. These runtime services are used by both the interpreter for bytecodes and code compiled by the native code compiler.

For such runtime services to function, all values created and manipulated by an OCaml program must conform to a specific internal representation. This is required by both compilers, the runtime services and, in the case of a bytecode program, the interpreter of the bytecodes, all of which must agree on the common internal representation. Since the internal representation is the same for both native-code-compiled and bytecode-compiled programs the same runtime services can be used for both provided that the compilers agree on the representation. For an OCaml program to use values from a further language, such as C, the foreign values must be marshalled to conform to this representation.

4.6.1.1 Target Application

This example concerns the writing of a comparison operator for OCaml values. To achieve this a function written in Nitro which inspects the OCaml data representation is exported as an external C function. This may then be called by any OCaml program. In the real runtime this function is written in C and the `=` operator is mapped to a call to the external C function. Note that if the entire runtime were re-written in Nitro, including the interpreter for bytecodes, then there would be no need to export it as an external C function, since the interpreter could call the Nitro function directly.

4.6.1.2 The data representation

This section details the data representation specific to the OCaml system. The roots of all values are stored in a one word sized storage location. These must be marked by the runtime as either a pointer into the heap or a long value. Pointers into the heap may be safely dereferenced, long values may not. For this purpose the least significant bit of every value is reserved as a flag to indicate whether the value can be dereferenced or not. The bit is set to `1` to indicate the value is a long value and set to `0` to indicate the value is a pointer into the heap. The runtime must make sure that long values are shifted to the left and marked as a long before being stored, and conversely shifted to the right before the value is manipulated, for example when two integers are added together. This means that integers are stored with one bit fewer than the word length of the machine.

A value that is a pointer can be dereferenced to obtain access to a *heap object*. All heap objects conform to the same structure. The first word is a header containing three pieces of information. The header describes how to interpret the remaining bytes of the heap object. For generic values the remaining bytes are an array of OCaml values.

The three pieces of information contained within the header word are as follows:

- A tag indicating the *kind* of heap object. This also indicates how to interpret the bytes/words that follow this header. There are a small number of predefined tag values to indicate a built-in type such as string, float or function closure. All other tag values are considered to represent a tagged union datatype or record type. In the case of a tagged union datatype the actual value of the tag is the tag of the union datatype.
- A colour field, used by the garbage collector.
- A length field which indicates how many bytes/words follow this header.

The structure of an OCaml value heap object header is shown in figure 4.8.

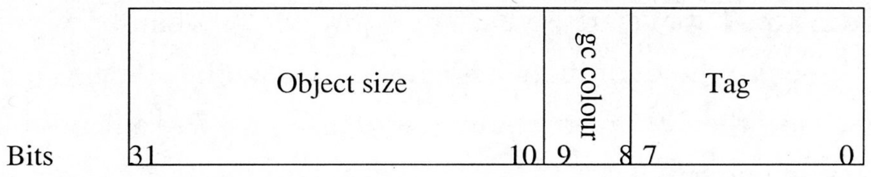


Figure 4.8: The structure of an OCaml heap object header.

There are several kinds of heap objects. Here is a description of a few of them which give the essence of the variation between the different kinds of heap objects.

Closure This is a function closure. It is treated much like a tuple, so there is an array of values that follows, except that the first value is a code pointer, and should not be accessed as a normal value.

String The bytes which follow are treated as characters. The length portion of the header indicates how many bytes there are.

Double The bytes which follow are to be interpreted as a double value. The length field of the header should always be the same.

Double array Similar to a double except that there is a number of double values to follow, that number being held in the length field of the header.

Generic Not one of the predefined tags, this represents a tagged union or record type. Such heap object headers are always followed by an array of OCaml values.

The important part is that the tag within the header describes the kind of values that follow the header in memory, how many there are and how to interpret them.

4.6.1.3 Type Definitions

In this section the Nitro definitions used to interface with the representation of OCaml values described above are given. The equivalent C definitions are then described. These are the definitions which are used within the OCaml runtime system because the OCaml runtime is written in C. The Nitro type definitions are given in Figure 4.9.

A point to note is the definition of the `Double` tag. The representation always stores the length of the value in the length field even when the length is fixed as with the `Double` tag. This means it can be used without checking the tag in the C code. It would have been possible to define the length field in the `Double` tag in the same way that was done for the other tags. However, because the double tag's length is fixed it can also be fixed within the constructor. This has the effect that the `Double` constructor cannot be used to create a double value with an incorrect length field.

If the double tag were a variable length field, then this could be used to create a double value with an incorrect length field. This is because the declared argument to the `Double` tag is a pair rather than a value array. It would still be impossible to make an incorrect access from within Nitro however an invalid value could be created which is then passed back to the current OCaml runtime. The runtime could in turn use the incorrect length field and overrun the

```
type gc_colour =
     Caml_white {0x000}
   | Caml_grey  {0x100}
   | Caml_blue  {0x200}
   | Caml_black {0x300}

type heap_object =
      Closure {247 with mask 0xFF;
               _   with mask 0x300 of gc_colour;
               _   (>> 10) of int;}

  (*
     In the Double tag we combine the tag field with the
     length field , because the length is always the same.
  *)
    | Double { 0x8FD with mask 0x8FF;
               _   with mask 0x300 of gc_colour;
             } precedes (int, int)

    | Tag { _ with mask 0xFF of int;
            _ with mask 0x300 of gc_colour;
            _ (>>10) of int.(len) }
            precedes ocaml_value array.(len)

and immediate ocaml_value =
      Ptr { 0 with mask 1} of heap_object
    | Long { 1 with mask 1} (>>. 1) of int
```

Figure 4.9: The Nitro type definitions for OCaml values.

argument to the double tag. Hence the method used of declaring the length as one value is preferable in this instance.

In contrast to the Nitro definitions, the C definitions are spread out between types and values. In particular creaction and inspection macros are defined separately from the type, and perhaps more significantly, separately from each other. For conciseness, included here is a small sample of the C definitions. The remainder can be found in the OCaml distribution available at `http://caml.inria.fr/ocaml/release.en.html` and the file `byterun/mlvalues.h` contains the relevant C definitions.

```
/* The type of ocaml values */
typedef long value;
```

These macros simply test whether a value is a pointer or a long value. Using these macros corresponds to matching against the `Ptr` and `Long` Nitro constructors matching the arguments with the underscore.

```
/* Longs vs blocks. */
#define Is_long(x)    (((x) & 1) != 0)
#define Is_block(x)   (((x) & 1) == 0)
```

These are the creation and access macros for long values. There are no corresponding macros for pointer values. To use a pointer value it should be first tested using the `Is_block` macro and then the original value is used. To use a long value it is first checked with the `Is_long` macro and then the `Long_val` macro is used to extract the represented integer. To create a long value the `Val_long` macro is used.

```
/* Conversion macro names are always of the form "to_from". */
/* Example: Val_long as in "Val from long" or "Val of long". */
#define Val_long(x)      (((long)(x) << 1) + 1)
#define Long_val(x)      ((x) >> 1)
```

4.6.1.4 The Equality function definitions

Given the two sets of definitions for the OCaml value representation this section now details two implementations of the equality operator. One is written in Nitro and the other in C. The test is for structural equality rather than physical equality. A physical equality test can simply determine if two values are exactly the same, a structural equality test must examine the structure of both values.

4.6.1.5 The C comparison function definition

The source code for the C version of the OCaml comparison function is shown below. This is taken from the OCaml distribution [11], I have added some comments and omitted some cases for the purpose of clarity.

```
/* Booleans are integers 0 or 1 */
#define Val_bool(x) Val_int((x) != 0)
#define Val_false Val_int(0)
#define Val_true Val_int(1)

static int compare_val (value v1, value v2)
{  if (v1 == v2)
       return Val_true ;
  /* If both values are longs, then they should be exactly
       the same and caught by the above case. This test
       also catches the case that one is a long and one is
       a block. */
  else if (Is_long (v1) || Is_long (v2))
       return Val_false ;
  /* We can now assume that both are block values, so we
       test the tag value, either they are not equal, in
       which case we fail or they are in which case we
       distinguish between the kinds of heap objects. */
  else {
     tag_t t1, t2;
     int size1, size2;

     t1 = Tag_hd(Hd_val(v1)) ;
     t2 = Tag_hd(Hd_val(v2)) ;
     size1 = Wosize_hd(Hd_val(v1)) ;
     size2 = Wosize_hd(Hd_val(v1)) ;
```

```
    if (t1 != t2)
      return Val_false ;

    /* We match against only one tag since we know
       both are the same */
    switch (t1) {
      case Closure_tag:
        /* Raises an ocaml exception */
        caml_invalid_argument("equal: functional value") ;
      case String_tag: {
        char * p1, * p2 ;

        p1 = (unsigned char *) String_val(v1) ;
        p2 = (unsigned char *) String_val(v2) ;

        return (Val_bool (compare_strings
                              (size1, p1, size2, p2))) ;
      }
      case Double_tag: {
        double d1 = Double_val(v1);
        double d2 = Double_val(v2);
        return (Val_bool (d1 == d2)) ;
      }
      default: /* Ordinary tagged value */ {
        int i ;
        for (i = 0; i < size1 && i < size2; i ++)
          if (compare_val (Field(v1, i), Field(v2, i))
                               == Val_false)
            return Val_false ;

        /* If all the fields are equal, return true */
        return Val_true ;
      } /* End of default case */
    } /* End of switch statement */
  } /* End of else command */
```

4.6.1.6 The Nitro comparison function definition

Figure 4.10 depicts the Nitro code used to define the OCaml values for **true** and **false** of the OCaml boolean type. In Nitro these values have the type `ocaml_value`.

```
let ocaml_true =  Ptr { Tag {0;0;1} Array.empty }
let ocaml_false = Ptr { Tag {0;0;0} Array.empty }
```

Figure 4.10: OCaml true and false in Nitro.

The Nitro code shown below defines comparison for OCaml values. The infix operator == is a library function in Nitro which defines physical equality.

```
let rec compare_val v1 v2 =
  if v1 == v2
  then ocaml_true
  else match v1, v2 with
         Ptr p1, Ptr p2 -> compare_heap_object p1 p2
       | _ ->
          (* Two equal Longs would be physically equal
             and hence caught by the if above *)
          ocaml_false
       end

and compare_heap_object o1 o2
  match o1, o2 with
    Closure _, _
  | _, Closure _ ->
     caml_invalid_argument("equal: abstract value")

  | String {_; _; len1} s1,
    String {_; _; len2} s2 ->
     let res = compare_strings (len1, s1, len2, s2) in
       Ptr { Tag {0;0; res} Array.empty }

  | Double (d1f, d1s), Double (d2f, d2s) ->
     if (d1f == d2f) && (d1s == d2s)
    then ocaml_true
    else ocaml_false

  (*
    This case is required by the compiler so that in the final
    case it knows that both arguments must be values created
    with the Tag pattern.
  *)
  | (String _ | Double _),
    (String _ | Double _) ->
    ocaml_false

  | Tag {t1; _; len1} a1, Tag {t2; _; len2} a2 ->
    if (t1 == t2) && (len1 == len2)
    then
      let rec match_args i =
        if i < #len1
        then
          let res = compare_val a1.(len1).[i]
                                a2.(len2).[i]
          in if res == ocaml_false
             then ocaml_false
             else match_args (i+1)
        else ocaml_true
      in match_args 0
    else ocaml_false

end
```

In order to be able to call this from an OCaml program as an external C procedure the definition is made with the **export** keyword.

```
export ocaml_compare = compare_val
```

4.6.1.7 Evaluation

Two implementations of an equality function for OCaml values, written in C and Nitro respectively have been defined in this chapter. This section evaluates the two solutions, including the type definitions and how they may help to write and maintain other such functions.

4.6.1.7.1 Safety It has been a design goal of the Nitro foreign data interface to remain type safe. This means that we cannot create a value in Nitro and access it from within Nitro at an incompatible type. This is important as it means we can use the control over data representation to code our own private data structures, perhaps optimising for speed or space. It also means that if we correctly code our interface then we cannot incorrectly access a foreign value, or create an invalid foreign value to return, though we can still cause the calling program to fail because it relies on some human maintained protocol.

In our example we cannot return to the OCaml runtime an invalid OCaml value. This means that for example the garbage collector will not fail because of some value which the Nitro code has returned. However the program may still fail since Nitro type definitions do not allow the encoding of OCaml type information, hence it would be possible to, for example return to the user program an OCaml double where an OCaml string was expected.

The C implementation is subject to all of these possibilities as well, and in addition there is no assurance that given a valid OCaml value the C procedure does not make an illegal access, or that it does not return an invalid OCaml value.

We say that a program is *value-safe*, if the program is type-safe given the condition that it is never supplied with an external value which does not conform to the representation expected by the interface to that external source. Furthermore if the external source is also value-safe then the combination of our program and the external source is type-safe. Nitro programs are value-safe and therefore if the interface, which is written in Nitro, is correct then the whole program is type-safe. An implementation in C is not checked by the compiler to be value-safe, since it may misuse any value supplied from the external source or return an invalid value back to the external source, which may in turn misuse it based on the false assumptions made by the interface.

Because Nitro implementations are value-safe an important class of programming errors can be detected by the type system. Our example above can still fail but only when the interface is wrong. When the interface states that we may return to the external source any OCaml value when in fact the external source expects an OCaml value of a specific OCaml type. If the external source does not make this assumption, that is, it assumes the value returned is an OCaml value but does not assume anything about its OCaml type then our program cannot fail, because our interface is therefore correct and the Nitro implementation and external source are both value-safe. In chapter 5 the type system is enhanced to support the definition of more accurate interfaces within Nitro and therefore more programs can be guaranteed to be type-safe. In particular the equality function defined in this chapter can be ensured to return only OCaml boolean values.

As was mentioned during the discussion of the C implementation, there is nothing to check that we have ensured a value is a block before dereferencing it and accessing the header. Likewise in the C implementation there is nothing to check that once a block header has been accessed the code does not access the block past the length given in the header. In the Nitro version it is simply not possible to use a `Long` value as a heap object.

When checking a generic tagged value the C implementation first checks that both blocks are the same length, and then accesses only up to that length. There is nothing to check that we perform the first check and nothing to make sure that we do not access past the end of either block. The Nitro implementation though, because of the way bare arrays are typed, checks each array access with its own length. It is therefore not possible to access memory erroneously past the end of either block.

4.6.1.7.2 Speed One of the reasons often given for choosing C as the implementation language for code perceived as low-level is efficiency. One source of efficiency is that it is possible to use reasoning that cannot be expressed within the type system to remove safety checks. As an example of one such check, consider that both versions check that the lengths of the arrays in two generic tagged values are equal. This check could be removed knowing as we do that the two values tested for equality must be of the same OCaml type. This means that if they have the same tag they must have the same number of arguments and hence length of array.

Removing such checks with only a human guarantee that it is safe to do so is naturally a dubious practice. In addition the extra type information about foreign values, retained by the Nitro type system, can allow optimisations that are not immediately obvious and certainly not easily maintained. For example, in our equality test, when accessing the argument value arrays of both heap objects a naïve Nitro compiler will insert array bounds checks. A smarter compiler however could use the type information and the equality check to remove the bounds checks. This means we save on as many bounds checks as the C version. Should a change occur which invalidates our assumption then the appropriate array bounds checks are re-inserted. For example if we remove the equality check on the array lengths, then the bounds check on the second array would be re-inserted.

After these optimisations have been applied the resulting order of checks is often not what a programmer would naturally write. I speculate that using such optimisations matching over the heap object type could be made more efficient than the corresponding C code. For an example of work on optimising pattern matching see [59].

4.6.1.7.3 Maintainability The Nitro definition of the OCaml values is entirely contained within type definitions. For the C version there are type and macro definitions. A change in the Nitro type definition automatically causes the compiler to flag up all portions of code dependent on the type that must also be updated. Because of the extra safety guarantees a change in the Nitro definition means that the rest of the code is checked for consistency. This is not true of the C version.

4.6.2 Ncurses Editor

This example is intended to show the expressiveness of the Nitro foreign interface. The editor itself is written in Nitro, and for the most part the Nitro code could be that of a high-level functional language. Combining the control over data representation with the benefits of the high-level features offered by Nitro we obtain a compromise solution in which we can implement the editor with suitable high-level constructs but do not pay the cost of expensive marshalling routines when interfacing with the legacy C coded library. In addition combining the interface described here with the OCaml interface described in the preceding section, one could implement the marshalling routines required for an OCaml interface to the legacy library.

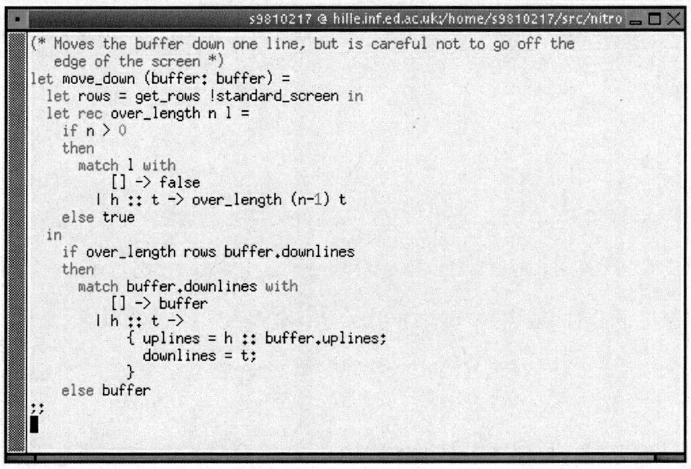
4.6.2.1 Background

Ncurses is a programming library providing an API which allows the programmer to write text-mode user interfaces in a terminal-independent manner. It also optimises screen changes, in order to reduce the latency experienced when using remote shells. Ncurses [60] stands for "new curses", and is a replacement for the discontinued 4.4BSD classic curses. There are many programs built using the ncurses library, including text editors, web browsers, instant messaging clients, package management tools and even integrated development environments.

Because the library is based upon an old library and has been around itself for many years, the library makes extensive optimisations for both space and time, relying on full control over data representation. This makes the library a good test for the Nitro foreign data interface.

4.6.2.2 The Nitro Editor

The *nitro editor* is a text editor specialised for use in writing Nitro programs. It highlights the syntax for Nitro programs and allows a basic but useful form of automatic indentation. There is also folding of function definitions (whereby a function is folded so that only the top-line is visible) and various other common keyboard commands such as "move to the end of the line". It uses the ncurses library for writing to the terminal but is entirely written in Nitro. There is no need for marshalling routines because the program is entirely within the type system for Nitro. The bindings to the ncurses library are of course reusable in other Nitro programs. A screenshot of the editor in action is shown in Figure 4.11.



```
(* Moves the buffer down one line, but is careful not to go off the
   edge of the screen *)
let move_down (buffer: buffer) =
  let rows = get_rows !standard_screen in
  let rec over_length n l =
    if n > 0
    then
      match l with
          [] -> false
        | h :: t -> over_length (n-1) t
    else true
  in
    if over_length rows buffer.downlines
    then
      match buffer.downlines with
          [] -> buffer
        | h :: t ->
            { uplines = h :: buffer.uplines;
              downlines = t;
            }
    else buffer
;;
```

Figure 4.11: . The ncurses Nitro editor in action.

4.6.2.3 The main type definitions

This section details the main type definitions which form the basis of the binding to the ncurses library.

4.6.2.3.1 Characters Here is the listing for the definition of the key codes used in the library. These are returned from functions such as `get_ch` which waits for the user to press a key and returns the key code representing it. Most of the key codes are defined as constants, the dots at the end represent that there are many more to follow. We could also write out all of the ASCII character codes as constants, however this has the disadvantage that there is no simple pattern which means, 'is an ASCII character'.

Under ncurses, if the key is an ASCII character code, then the ASCII code is given in the least significant byte and the rest of the word value is zero. Hence as our pattern we reverse the mask that selects only the least significant byte and we have an argument that matches the whole value that is a `char`. Note that it is a Nitro `char` and not a C `char`. All of the other key codes have no arguments since they are constants.

```
type immediate  key_code  =
     KEY_ASCII {0 with mask -255} of char
   | KEY_CODE_YES {256}
   | KEY_MIN {257}
   | KEY_BREAK {257}
 ...
```

Giving a mask for the KEY_ASCII constructor means that we can write functions such as the one shown in Figure 4.12 which removes white space and control keys from a list of key presses. Had we given an individual constructor to each character we would need a case here for every single character.

```
let rec remove_whitespace chars =
 match chars with
   [] -> []
 | (KEY_ASCII '\n') :: rest
 | (KEY_ASCII '\t') :: rest
 | (KEY_ASCII ' ')  :: rest -> remove_whitespace rest

 | ( KEY_ASCII _ as k) :: rest -> k :: (remove_whitespace rest)

 | _ -> remove_whitespace rest
```

Figure 4.12:

4.6.2.3.2 Windows Windows are a basic part of the ncurses library. When interfacing with the library from C we are provided with several macros to extract information from the packed values which form the representation of a window.

A window is itself represented by a C **struct**, which we can map well with a Nitro record. The first element of the record is the current x and y coordinates the second is the maximum x and y coordinates.

```
type ncurses_window   = { current_y_x : xy_coord ;
                          maximum_xy : xy_coord ;
                        }
```

However the individual values are mostly packed into as few words as possible. As an example x and y coordinates are packed into a single word. On a 32 bit word machine this means that the two most significant bytes form the X coordinate and the two other bytes form the Y coordinate. Hence we require a way to extract these two values from the one word. We use an **immediate** type with a single custom constructor. The custom constructor defines two tag arguments. Just as the custom constructors for the OCaml heap object headers had to extract multiple values from within a word here we do the same thing. In the case of the `x` coordinate, because we are taking the most significant bytes, we must shift the value two bytes to the right to obtain the correct value. This was the same operation performed to get the true length value from the length portion of the header.

```
type immediate xy_coord   =
     YX_coord { _ with mask 65535   >> 0  of int
                _ with mask -65536 >> 16 of int }
```

4.6.2.3.3 Attributes Attributes are properties that control how characters are printed to the terminal. They can be attached to text or turned on and off for all characters printed to the screen. All but one of the attributes are simple on-off attributes that are either true or false, such as whether to print characters with an underline, or in bold. There is a single more complicated attribute which is the colour which can take one of two hundred and fifty five different values. Because all the attributes are so simple rather than make them all take up space the ncurses library stores all the attributes in one thirty-two bit value. There is enough space left over for a one byte character value. This means that on a machine with a word length of at least thirty-two bits we can store in a register a character to be printed to the screen and all the attributes with which to print it.

This could be represented using a single custom constructor with multiple tag arguments most of which are one byte long. However a more readable way is to give a separate constructor for each of the attributes. Using such a definition it means that subsequent functions operating over attributes can be written without the knowledge of how the attributes are represented. This style is also more tolerant of layout changes in that function definitions are less likely to require modification.

The following type definition describes the layout detailed above. This is then improved upon to allow more convenient testing for the absence of a particular attribute. The first style uses **immediate** constructors to access each of the individual attributes. There is also a constructor to access the character value stored in an attribute word. It is also simple to define a constructor to test for an attribute word in which all attributes are switched off – the `Normal` constructor is defined below for this purpose. The `||` after the custom tags are tag operations which ensure that the relevant bit is set after construction.

```
type immediate  t_attribute  =
      Char_value {_ with mask 255 of char}
    | Colour     {_ with mask 65280 of int}            of t_attribute

    | Normal     {0 with mask -256}                    of t_attribute
    | Standout   {65536 with mask 65536}   || 65536    of t_attribute
    | Underline  {131072 with mask 131072} || 131072   of t_attribute
    | Reverse    {262144 with mask 262144} || 262144   of t_attribute
    | Blink      {524288 with mask 524288} || 524288   of t_attribute

... Several more similar constructors
```

Note that all of the attribute constructors overlap each other. There are many attribute word values that could match two or more of these constructors. This is allowed by the Nitro compiler because all of these overlapping constructors have the same argument type, namely `t_attribute`. Even the `Colour` constructor is given a `t_attribute` argument, only the `Char_value` constructor is given no type. This means that all values of type `t_attribute` must have ultimately been created with the `Char_value` constructor and hence must have a character in the least significant byte. The compiler also insists that none of the other constructors overlap the `Char_value` or `Colour` constructors, because they have immediate arguments that could be changed otherwise.

Because the argument type of the attribute constructors is itself a `t_attribute` we can test for multiple attributes at once using nested patterns, for example.

```
let is_blink_and_underline attrib =
match attrib with
  Blink (Underline _) -> true
| _                   -> false
end
```

$: t\_attribute \rightarrow bool$

Note that it does not matter in which order the constructors were originally applied in order to obtain the value being tested or indeed if any constructors were applied at all because of course the value being matched against may have been created by the ncurses library itself.

There is still one significant drawback to this approach, and that is that testing for the absence of a specific attribute is laborious at best. One possibility is to test for its presence and then matching all values in a case below. As in this code fragment:

```
let is_not_blinking attrib =
match attrib with
  Blink _ -> false
| _       -> true
end
```

$: t\_attribute \rightarrow bool$

A better way is to give a corresponding negative constructor for each of the attributes. Here is the updated `t_attribute` type.

```
type immediate t_attribute  =
    Char_value  {_ with mask 255 of char}
  | Colour      {_ with mask 65280 of int}
  | Normal      {0 with mask -256}                     of t_attribute
  | Standout    {65536 with mask 65536}   ||  65536    of t_attribute
  | NoStandout  {0 with mask 65536}       && -65537    of t_attribute
  | Underline   {131072 with mask 131072} ||  131072   of t_attribute
  | NoUnderline {0 with mask 131072}      && -131073   of t_attribute
 ... Several more similar constructors
```

We have shown with the example of Ncurses attributes that a common C idiom, namely that of using bit masks as flags, can be imitated using the Nitro foreign data interface.

With these three type definitions we have covered the main functionality of the ncurses library. Most of the rest of the interface consists of external function declarations. It has been shown that we can represent the data structures of the ncurses library in Nitro. In particular three common forms used in C libraries;

- Enumeration types often done using integers and the preprocessor as with the `key_code` example.
- Combining smaller types together into one word without losing information.
- Bit masks used to hold together flags of various properties within one value that can be passed around.

4.6.2.4 A Note on Portability

Some of the definitions in this example have been necessarily restricted to a particular architecture or set of architectures. This unfortunately arises whenever the user is given access to the concrete data representation. When using a high-level language the implementor of the marshalling routines required for a legacy library interface must also take portability into consideration. However this is commonly done for them by the library implementor in the form of macros and defined constants. For Nitro, a preprocessor could be written to help with portability and in some cases the C preprocessor itself could be used. We have already seen a case where we have used the Nitro defined constant `Wordsize` to implement a tagged data type. More such constants and macros could be defined to help with portability.

4.6.2.5 Conclusions

This section has described an interface to a legacy C library, allowing direct access from Nitro, a type-safe functional language. Since this particular legacy C library utilises common idioms used in C data representation to optimise for both space and time we are given confidence in the expressiveness of the foreign data interface of Nitro. While it is doubtless that there exist some representations (for example see Section 7.3.1.1) that cannot currently or perhaps ever be represented directly in a type-safe language, the examples in this chapter provide encouragement to map as much as is possible. For those representations that remain outside of the scope of Nitro, there is no choice but to write a marshalling routine, however such representations can often be considered questionable and may eventually become obsolete.

4.7 Conclusions

This chapter has presented example uses of the Nitro foreign data interface. It has been shown that a large degree of control over data representation need not require that we sacrifice safety guarantees.

We have also shown that interfacing with foreign data from a type-safe language need not require that we write marshalling routines to package up the foreign data into our own format. Where this is desirable we can at least write those routines in our own language rather than resort to a (perhaps third) low-level implementation language.

The two examples have shown an ability to manipulate values from the higher-level OCaml language, and from the lower-level C language. Combining the type definitions from both we can provide an OCaml interface to the ncurses library. In this way we can use Nitro— a type-safe functional language — to provide the marshalling routines necessary to translate ncurses values into the common internal representation of the OCaml environment.

There is a web demonstration of a version of Nitro that includes a traditional type inference scheme augmented with typing for bare arrays and the foreign data facilities described above. This can be found at `http://homepages.inf.ed.ac.uk/s9810217/foreign_data_demo.html`.

Chapter 5

Delayed Typing

5.1 Introduction

In the previous chapter Nitro was given a foreign data interface which utilised the power of tagged union data types by allowing the user to provide representation requirements. This means that the layout of data in memory can be controlled by the user. The type system still maintained that the user could not make illegal accesses to any data structure and therefore the benefits of increased security and error detection by the compiler were retained. The type system however lacked the ability to allow sub-typing relations and this was noted to be a serious drawback as it suppressed the ability of the programmer to encode foreign type information within the types of the Nitro code. For abstraction-level code this meant that the interfaces to the languages for which the abstraction is being provided were coarser than desired.

This chapter details a novel typing scheme which can incorporate the foreign data type additions detailed in the previous chapter and also allow the inference of sub-typing constraints. The resulting type system also admits a general scheme for inferring type annotations such as those which describe the effects which the evaluation of an expression can have. Effects are such actions as the raising of an exception or the accessing of a region in memory. As something of a bonus, the new typing scheme allows the typing of more programs meaning that some of the *slack* of previous typing schemes can be included. The *slack* of a typing scheme is the set of programs which are safe programs but which the typing scheme will reject [61]. This is because for a safe statically-typed language, the typing scheme admits a conservative approximation to the set of all safe programs.

This chapter begins with a review of the need to allow sub-typing within the type system of an abstraction-level programming language. The delayed typing scheme is then introduced and a formal static semantics in the form of inference rules are presented for a basic lambda-calculus. This is then extended with typing constraints, record types, a sub-typing relation and side effecting expressions through the use of mutable record fields. Finally exception raising and catching capabilities are added to the lambda-calculus and the inference of exception effect annotations is detailed.

An algorithm to infer types under the delayed typing scheme is presented and some properties of the delayed typing scheme are shown in section 5.13.

In section 5.14 the delayed typing scheme is then compared with the traditional Hindley-Milner typing scheme and several other typing schemes which augment Hindley-Milner.

The chapter concludes with a description of how the delayed typing scheme was incorporated into the Nitro programming language and some conclusions.

5.2 Motivation

In this section the motivations for the development of the delayed typing system are reviewed.

In the previous chapter it was noted that the lack of sub-typing impeded the programmer's ability to encode foreign type information. Returning a value from the `compare` function, it was possible for the type system to ensure that a valid `ocaml_value` was returned. However it would be desirable to ensure that the value returned was either `ocaml_true` or `ocaml_false`.

This would not only increase the confidence in the correctness of the `compare` function, it would also facilitate the writing of the OCaml compiler. On encountering the expression `if x == y then 1 else 0` the compiler must emit a call to the Nitro defined `compare` function and instructions which match the result of that call to the OCaml representation of `true` and/or `false`. Recall that an OCaml `boolean` value is represented as a boxed tagged value, this means that the value is in fact a pointer to a header in which the tag portion distinguishes between true and false. This is shown in Figure 5.1. The upper smaller box is the pointer, distinguished from a long value by the zero in the least significant bit. This pointer points at a block, however the block has zero length after the header. The value is entirely defined within the Tag portion of the header which may take one of two values to represent either `true` or `false`. The other fields of the header are the garbage collector colour whose value we do not know, and the object size, that is the number of values which follow the header, which we know to be zero.

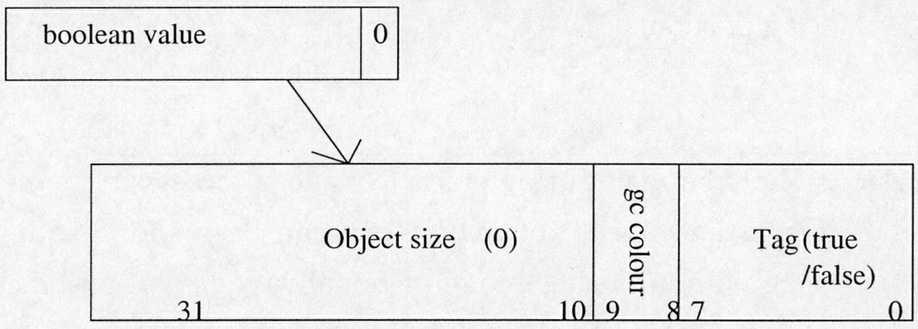


Figure 5.1: The structure of an OCaml boolean value.

The instructions emitted by the compiler to perform the comparison depend on the guarantees made by the `compare` function. In particular what those instructions must check for. The differences are summarised in Table 5.1.

Without sub-typing, the programmer could of course define a second type `ocaml_boolean` with a similar but restricted definition to that of `ocaml_value`. However in doing so the programmer prevents any value of type `ocaml_boolean` from being used wherever it is possible to use an `ocaml_value`. For example it would not be possible to call the defined `compare` function with an `ocaml_boolean` value

Guarantee	*Check*	*Possible Errors*
No guarantees	The compiler assumes it is an `ocaml_value` and may check if it is safe to dereference to obtain the tag part. It then must check if the tag portion is equal to true or false.	We may make an illegal access to memory, or if the value is not equal to true, the else branch will be taken, but the value may not be equal to false either.
Is an `ocaml_value`	The compiler may check to see if it is safe to dereference the value to obtain the tag portion.	We now cannot make an illegal access to memory. However unless there is a check, if the tag is not equal to true it may also not be equal to false.
Is an `ocaml_boolean`	The compiler can dereference the value to obtain the tag without a check to ensure it is safe. If the value is not equal to true, then it must be equal to false.	No illegal access to memory can be made. Additionally we cannot mistake a non-boolean value which is not equal to true, to be equal to false, because the value is definitely a boolean.

Table 5.1: Checks inserted on a call to the Nitro defined compare function

as either of the arguments. Worst still, if the type definition for `ocaml_value` is changed then the programmer must remember to update the type definition for the separate `ocaml_boolean` type.

5.3 Delayed Typing Formalisation

This section details a formal semantics for the delayed typing scheme. It begins with a basic lambda-calculus, this is then extended to include record expressions and a sub-typing relation and then further extended to allow side-effecting expressions through the use of mutable record fields. Finally, exception raising and catching facilities are added and the delayed typing scheme is augmented with the ability to infer accurate exception annotations on the types.

5.3.1 Delayed Types

In this section a delayed type is briefly explained before a set of formal inference rules which define the delayed typing scheme over a basic lambda-calculus are given.

A delayed type is a type containing an expression which is said to be waiting to be typed. Because the typing of an expression is delayed, it can be given a type accurate to the situation in which it is used. When an expression is bound to an identifier, the identifier can be given a delayed type, the expression associated with the delayed type can then be typed differently according to the separate uses of the bound identifier. In this way a delayed type is an efficient representation of a set of types which are appropriate for the given expression (and hence the identifier to which it is bound).

A delayed type is written

$\tau \quad := \quad [[e]]$

the expression inside the delayed type has a special syntax which will be shown in the section 5.12.1, for now these are expressions which may contain as any sub-expression a type. Here is an example:

$[[\textbf{fun}\ r \rightarrow (r, r.lab, \textbf{int})]]$

This represents the set of all types which are an arrow type, where the argument type is a record type containing the field *lab* and the return type is a three tuple consisting of, the original argument type, the type associated with the *lab* field and **int**. Examples of the set include

$(\{lab : \textbf{int}\ ;\} \ \rightarrow\ (\{lab : \textbf{int}\ ;\}, \textbf{int}, \textbf{int}))$

$(\{lab : (\textbf{bool}\ \rightarrow\ \textbf{bool})\ ; lab2 : \textbf{bool}\ ;\} \ \rightarrow$

$\quad (\{lab : (\textbf{bool}\ \rightarrow\ \textbf{bool})\ ; lab2 : \textbf{bool}\ ;\}, (\textbf{bool}\ \rightarrow\ \textbf{bool}), \textbf{int}))$

Since a delayed type is equivalent to a set of types the inference rules make no mention of delayed types but refer only to sets of types. Each expression can be given a set of appropriate types. An algorithm for inferring delayed types is given in section 5.12, this also contains the syntax of delayed types. Since sets of types, and in particular infinite sets of types, are awkward to display to the programmer, an implementation of a delayed typing scheme for a programming language such as Nitro would display delayed types as the representation for a set of types.

5.4 The basic lambda calculus

In this section a simple lambda calculus is defined and the static semantics for a delayed typing scheme over that lambda calulus are given. This basic lambda calculus represents a subset of the core Nitro defined in Chapter 3. In the following sections this basic lambda calculus will be augmented with features found in Nitro.

5.4.1 Syntax

The syntax for the subset of expressions is given in Figure 5.2 and the syntax for type in Figure 5.3

Notice that the syntax for types allows types to contain type schemes. This allows the inference of a type such as $((\forall('a).'a\ \rightarrow\ 'a)\ \rightarrow\ \textbf{int})$ which is the type of a function which accepts as its first argument a polymorphic function.

$$\begin{array}{rcl} e & := & c \\ & | & x \\ & | & \textbf{let}\, x \;=\; e_1 \;\textbf{in}\; e_2 \\ & | & e_1\; e_2 \\ & | & \textbf{fun}\, x \rightarrow e \end{array}$$

Figure 5.2: The syntax of expressions in the basic lambda calculus

$$\begin{array}{rcl} \tau & := & \forall('a_1, \ldots 'a_n).\tau \\ & | & \textbf{int} \\ & | & \textbf{bool} \\ & | & (\tau_{arg} \;\rightarrow\; \tau_{res}) \\ & | & 'a \end{array}$$

Figure 5.3: The syntax of types in the basic lambda calculus

5.4.2 Typing Rules

Please note, the character $\overline{\tau}$, as distinct from τ is used to denote a set of possible types. The rules make use of a set notation where a generic member of the set is given on the left hand side of the vertical bar and on the right hand side the conditions on that generic member. So for example $\{(\tau \;\rightarrow\; \tau_1) \mid \tau \in \overline{\tau}\}$ would represent the set of all types which are arrow types such that the argument type is in the set of types $\overline{\tau}$.

$\boxed{C \vdash e \Rightarrow \overline{\tau}}$

The first three rules apply to all expressions. They allow a typing deduction to refine the set of types deduced for an expression.

$$\frac{C \vdash e \Rightarrow \{\tau_1\} \qquad \tau = inst(C, \tau_1)}{C \vdash e \Rightarrow \{\tau\}} \tag{110}$$

$$\frac{C \vdash e \Rightarrow \overline{\tau}_1 \qquad \overline{\tau} \subset \overline{\tau}_1}{C \vdash e \Rightarrow \overline{\tau}} \tag{111}$$

$$\frac{C \vdash e \Rightarrow \overline{\tau} \qquad \tau \in \overline{\tau} \qquad {'a} \text{ not free in } C}{C \vdash e \Rightarrow \overline{\tau} \cup \{\forall({'a}).\tau\}} \tag{112}$$

The remaining rules each apply to one grammatical form of expression.

$$\frac{\mathcal{CON}(c) = \overline{\tau}}{C \vdash c \Rightarrow \overline{\tau}} \tag{113}$$

$$\frac{C(x) = \overline{\tau}}{C \vdash x \Rightarrow \overline{\tau}} \tag{114}$$

$$\frac{\overline{\tau} = \{(\tau_1 \ \rightarrow \ \tau_2) \mid C_x[x \mapsto \{\tau_1\}] \vdash e \Rightarrow \overline{\tau}' \wedge \tau_2 \in \overline{\tau}'\}}{C \vdash \textbf{fun}\ x \rightarrow e \Rightarrow \overline{\tau}} \tag{115}$$

$$\frac{C \vdash e_1 \Rightarrow \overline{\tau}_1 \qquad C \vdash e_2 \Rightarrow \overline{\tau}_2}{C \vdash e_1\ e_2 \Rightarrow \{\tau \mid (\tau_2 \ \rightarrow \ \tau) \in \overline{\tau}_1 \wedge \tau_2 \in \overline{\tau}_2\}} \tag{116}$$

$$\frac{C \vdash e_1 \Rightarrow \overline{\tau}_1 \qquad C_x[x \mapsto \overline{\tau}_1] \vdash e_2 \Rightarrow \overline{\tau}_2}{C \vdash \textbf{let}\ x \ = \ e_1\ \textbf{in}\ e_2 \Rightarrow \overline{\tau}_2} \tag{117}$$

5.5 Notes

The *inst* function used in rule 110, substitutes any bound type variable for a type in the body of a type scheme. Hence the type $\forall({'a}).({'a} \ \rightarrow \ \tau)$ can become the type $(\tau_1 \ \rightarrow \ S(\tau))$ where $S = [{'a} \mapsto \tau_1]$ However the substitution must rename any of the bound type variables occurring in τ as appropriate to avoid variable capture on any of the free variables in τ_1.

The subsumption rule 111 allows a smaller set of types to be inferred for a given expression. In particular this rule can be used to allow a singleton set of types to be inferred. This ability will be used in the following section to enforce explicit type constraints given by the programmer.

The application rule 116, may infer types for the argument expression which are not applicable to the function. Additionally for the function expression there may be many types in the set inferred for it that accept none of the types in the set inferred for the argument expression. In general there may be many types in the sets which are inferred for both the function and the argument expression which are inappropriate for the application expression. However the rule will not allow those inappropriate types to form part of any set inferred for the whole application expression.

5.6 Adding in Type Constraints

In this section the basic lambda calculus is extended by allowing typing constraints to be applied to both expressions and binding locations for identifiers. Note that typing constraints are single types and not sets of types.

5.6.1 Additional Syntax

The additional syntax is straightforward and similar to most variants of the SML language. The brackets around the type constrained argument the abstraction expression are optional and provided here for clarity.

$$\begin{array}{lcl} e & := & e : \tau \\ & \mid & \textbf{fun}\ (x : \tau) \rightarrow e \\ & \mid & \textbf{let}\ x : \tau\ =\ e_1\ \textbf{in}\ e_2 \end{array}$$

5.6.2 Additional Typing Rules

The additional typing rules restrict the set of types inferred for each constrained expression (or identifier) to a single type corresponding to the given type constraint. The sub-expressions may in general have a larger set of types inferred for them, but this set can always be reduced to the singleton set containing only the constraint type by using the subsumption rule 111, assuming that the constraint type is in the set in the first place.

$$\frac{C \vdash e \Rightarrow \{\tau\}}{C \vdash e : \tau \Rightarrow \{\tau\}} \qquad (118)$$

$$\frac{C \vdash e_1 \Rightarrow \{\tau\} \qquad C_x[x \mapsto \{\tau\}] \vdash e_2 \Rightarrow \overline{\tau}}{C \vdash \textbf{let}\, x : \tau \;=\; e_1 \;\textbf{in}\; e_2 \Rightarrow \overline{\tau}} \tag{119}$$

$$\frac{C_x[x \mapsto \{\tau\}] \vdash e \Rightarrow \overline{\tau}}{C \vdash \textbf{fun}\;(x : \tau) \rightarrow e \Rightarrow \{(\tau \;\rightarrow\; \tau_1) \mid \tau_1 \in \overline{\tau}\}} \tag{120}$$

5.7 Adding Recursion

Adding recursion to the delayed typing scheme is not as trivial as with a strictly unified system. To ease the job of the inference algorithm all recursively defined values must be constrained with a single type. Additionally, as in the core Nitro language, a syntactic restriction which restricts the initialising expression to be a function abstraction is used to avoid incomputable cyclic values such as:

$\textbf{let rec}\; x : \textbf{int} \;=\; x \;\textbf{in}\; x$

5.7.1 Additional Syntax

The additional syntax is simply the **let rec** expression. There are no additional types required.

$e \;:=\; \textbf{let rec}\; x : \tau \;=\; \textbf{fun}\; y \rightarrow e_1 \;\textbf{in}\; e_2$

5.7.2 Additional Typing Rules

The typing rules do not allow a set larger than the singleton set to be inferred for the initialising expression. In any case this must match with the type constraint which must be a single type. This means that there is no polymorphic recursion.

$$\frac{C_x[x \mapsto \{\tau\}] \vdash (\textbf{fun}\; y \rightarrow e_1) \Rightarrow \{\tau\} \qquad C_x[x \mapsto \{\tau\}] \vdash e_2 \Rightarrow \overline{\tau}}{C \vdash \textbf{let rec}\; x : \tau \;=\; (\textbf{fun}\; y \rightarrow e_1) \;\textbf{in}\; e_2 \Rightarrow \overline{\tau}} \tag{121}$$

5.8 Adding Records

The addition of record expressions and types will enrich the type system such that in the following section sub-typing can be usefully added. Recall that the major goal of the delayed typing scheme was to allow for sub-typing on record and tagged union types.

5.8.1 Additional Syntax

The additional syntax is again straightforward. It is a departure from SML record syntax for field access in which the OCaml convention of using the dot notation is used.

$$\begin{array}{lll} e & := & e.field \\ & | & \{label = e\ ;^{+}\ \} \end{array}$$

Types are updated with:

$$\begin{array}{lll} \tau & := & \{label : \tau\ ;^{+}\ \} \end{array}$$

5.8.2 Additional Typing Rules

The rule for field access makes use of the function $\mathcal{F}$, this takes two arguments: a type and a field label. If the given type is a record type containing a field with the given label then the type which the field is mapped to is returned. Otherwise the function fails.

The rule for field access show here, allows for the possibility that all of the types in the set inferred for the record expression contain more than one field. In the next section sub-typing between record fields is introduced such that this provision is not required, since any type could be reduced to a type containing only the accessed field label. Because those rules have not yet appeared in this section the rule for field access allows for larger record types.

$\boxed{C \vdash \{fields\} \Rightarrow \overline{\tau}}$

$$\frac{C \vdash e \Rightarrow \overline{\tau} \qquad \langle C \vdash \{fields\} \Rightarrow \overline{\tau}_1 \rangle}{C \vdash \{lab = e\ ; \langle fields \rangle\} \Rightarrow \{lab : \tau\ ; \langle fieldecs \rangle \mid \tau \in \overline{\tau} \langle \wedge \{fieldecs\} \in \overline{\tau}_1 \rangle\}} \quad (122)$$

$$\frac{C \vdash e \Rightarrow \overline{\tau}}{C \vdash e.lab \Rightarrow \{\mathcal{F}(\tau, lab) \mid \tau \in \overline{\tau}\}} \tag{123}$$

5.9 Adding Sub-typing

In this section a sub-typing relation is defined over record types and this relation extends naturally to types containing record types, including other record types.

5.9.1 Additional Typing Rules

The rules which follow define the sub-typing relation <: where $\tau_1 <: \tau_2$ denotes that τ_1 is a sub-type of the type τ_2.

$$\frac{\tau_1 = \tau_2}{\tau_1 <: \tau_2} \tag{124}$$

$$\frac{\mathcal{F}(\tau, label) <: \tau_1 \qquad \langle \tau <: \{fields\} \rangle}{\tau <: \{label = \tau_1 \,; \langle fields \rangle\}} \tag{125}$$

The sub-typing relation extends to types containing record types, currently the only such type is the arrow type. Here is the sub-typing rule for arrow types, notice that the parameter types are switched over because they are in the contra-variant position.

$$\frac{\tau_3 <: \tau_1 \qquad \tau_2 <: \tau_4}{(\tau_1 \rightarrow \tau_2) <: (\tau_3 \rightarrow \tau_4)} \tag{126}$$

Finally there is a subsumption rule for sub-typing, this allows the set of types which may be inferred for an expression to be increased by allowing a super-type of any of the types otherwise in the set.

$$\frac{C \vdash e \Rightarrow \overline{\tau}}{C \vdash e \Rightarrow \{\tau \mid \tau_1 \in \overline{\tau} \wedge \tau_1 <: \tau\}} \tag{127}$$

5.10 Adding Side Effects

In this section side-effects are added to the language through the use of mutable record fields.

5.10.1 Additional Syntax

$$
\begin{array}{lll}
e & := & \{\langle\textbf{mutable}\rangle\ \mathit{label} = e\ ;^{+}\ \} \\
\tau & := & \{\langle\textbf{mutable}\rangle\ \mathit{label} : \tau\ ;^{+}\ \}
\end{array}
$$

The additional syntax to accommodate destructive update for record fields is given by the following grammar. A *unit* expression and a unit type are added. The unit type is the type of a side-effecting expression that produces no result, such as a record field update. A unit expression can be used as the other case when such an action is performed conditionally.

$$
\begin{array}{lll}
e & := & e_1.\mathit{label} \leftarrow e_2 \\
 & | & () \\
\tau & := & ()
\end{array}
$$

5.10.2 Additional Typing Rules

Note that here the field expression can only be of one type, and hence the inference of any expression has to choose one type for all occurrences of the record field.

$$
\frac{C \vdash e \Rightarrow \{\tau\} \qquad \langle C \vdash \{\mathit{fields}\} \Rightarrow \overline{\tau}_1\rangle}{\begin{array}{c} C \vdash \{\textbf{mutable}\ \mathit{lab} = e\ ; \langle\mathit{fields}\rangle\} \Rightarrow \\ \{\{\textbf{mutable}\ \mathit{lab} : \tau\ ; \langle\mathit{fieldecs}\rangle\}\langle|\ \{\mathit{fieldecs}\} \in \overline{\tau}_1\rangle\}\end{array}} \tag{128}
$$

$$
\frac{C \vdash e_1 \Rightarrow \{\textbf{mutable}\ \mathit{lab} : \tau\ ;\} \qquad C \vdash e_2 \Rightarrow \{\tau\}}{C \vdash e_1.\mathit{lab} \leftarrow e_2 \Rightarrow \{()\}} \tag{129}
$$

5.10.3 Sub-typing with side-effects

The rules for sub-typing of record expressions must be modified with respect to the addition of the possibility for side-effecting expressions. The main update is that a mutable record field is now in the non-variant position. This means that a record type containing a mutable field is a sub-type of another such record type, only if their mutable fields have the exact same type, rather than, as with immutable record fields, the first being a sub-type of the second.

Here is the additional sub-typing rule for mutable record fields:

$$\frac{\textbf{mutable}\ \tau_1 = \mathcal{F}(\tau, label) \qquad \langle \tau <: \{fields\} \rangle}{\tau <: \{\ \textbf{mutable}\ label = \tau_1\ ; \langle fields \rangle \}} \tag{130}$$

5.11 Adding Exceptions

As well as allowing a sub-typing relation in the presence of type inference one of the goals of the delayed typing scheme was to allow the inference of accurate exception annotations. In this section, exception raising and catching abilities are added to the language and exception annotations are added to the types. The inference rules are then updated such that the types within the sets of types inferred for each expression contain annotations indicating the set of exceptions that may be raised by that expression.

5.11.1 Additional Syntax

A type now has attached to it a set of exceptions that the expression associated with the type may raise. A set of exceptions is written as: $[E_1; E_2; ...; E_n]$ The greek letter ξ will be used to range over sets of exceptions.

In order to allow the raising and catching of exceptions the syntax for expressions is extended by:

$$\begin{array}{lll} e & := & \textbf{raise}\ E \\ & | & \textbf{try}\ e_1\ \textbf{with}\ E \rightarrow e_2 \end{array}$$

The syntax of types must be updated:

$$\begin{array}{lll} \tau & := & \tau[\xi] \\ & | & \textbf{any} \end{array}$$

The new type **any** indicates that the expression can be thought of as having any type. This is the type given to an expression which is guaranteed to raise one of a set of exceptions. The value obtained by evaluating such an expression can be used in any context, since that value will never be obtained and hence its use will never be evaluated. Where a type occurs without an exception annotation this is assumed to mean the empty set of exceptions, that is, to indicate that no exceptions will be raised.

5.11.2 Additional Typing Rules

The important rules for the typing of exceptions are those which deal directly with the additional expression forms. These are:

$$\frac{}{C \vdash \textbf{raise}\ E \Rightarrow \{\textbf{any}[E]\}} \tag{131}$$

$$\frac{C \vdash e_1 \Rightarrow \overline{\tau}_1 \qquad C \vdash e_2 \Rightarrow \overline{\tau}_2}{C \vdash \textbf{try}\ e_1\ \textbf{with}\ E \rightarrow e_2 \Rightarrow \overline{\tau}_3 \cup \overline{\tau}_4} \tag{132}$$

where $\overline{\tau}_3 = \{\tau[\xi] \mid \tau[\xi] \in \overline{\tau}_1 \wedge E \notin \xi\}$
and $\overline{\tau}_4 = \{\tau[(\xi' \backslash \{E\}) \cup \xi''] \mid \tau[\xi'] \in \overline{\tau}_1 \wedge \tau[\xi''] \in \overline{\tau}_2\}$

In rule 132 the union of two sets of types may be inferred for a **try** expression. The first set $\overline{\tau}_3$ is the set of types which can be inferred for the expression e_1 which indicate that it cannot raise the exception E. Any of the types in this set may be inferred for the whole **try** expression regardless of which types (if any at all) may be inferred for the expression e_2. Because the expression e_1 will not raise the exception E, the handler expression e_2 will not be evaluated and hence its typing need not influence the typing of the **try** expression.

The second set $\overline{\tau}_4$ is essentially the set of types which may be inferred for both expressions e_1 and e_2, since the result of the **try** expression could potentially be the result of evaluating either of these two sub-expressions. The exception annotations on the types in the set $\overline{\tau}_4$ are modified to reflect the semantics of the **try** expression. If, ignoring the outermost exception annotations, it is possible to infer τ for both sub-expressions then τ is in the set $\overline{\tau}_4$. The exception annotation attached to τ in the set $\overline{\tau}_4$ is the union of the sets of exceptions inferred for both sub-expressions except that if E only occurs in the exceptions associated with e_1 then it is omitted.

An additional rule, Rule 133, allows the inference of any type for an expression which will certainly raise one of the exceptions in the given set. Usually such an expression will be combined with other expressions as part of a choice. Therefore it must be possible to infer the type of the other choices for the expression which will always raise the exception.

$$\frac{C \vdash e_1 \Rightarrow \{\mathbf{any}[\xi]\}}{C \vdash e_1 \Rightarrow \{\tau[\xi]\}} \tag{133}$$

Another way to allow this would be to change the rule for the **raise** expression to be:

$$\frac{}{C \vdash \mathbf{raise}\ E \Rightarrow \{\tau[E]\}}$$

However in the next section the rules for the typing of other expressions will be updated which take advantage of the fact that the type system allows a type to distinguish between an expression which will always raise one of a set of expressions (**any**$[\xi]$) and one which may raise one of a set of expression but may also evaluate to a value ($\tau[\xi]$).

5.11.3 Updated Typing Rules

All of the previous typing rules must be updated to account for exception sets. Here is the new rule for application.

$$\frac{C \vdash e_1 \Rightarrow \bar{\tau}_1 \qquad C \vdash e_2 \Rightarrow \bar{\tau}_2}{C \vdash e_1\ e_2 \Rightarrow \{\tau[\xi] \mid (\tau_2 \rightarrow \tau[\xi'])[\xi''] \in \bar{\tau}_1 \wedge \tau_2[\xi'''] \in \bar{\tau}_2\}} \tag{134}$$

where $\xi = \xi' \cup \xi'' \cup \xi'''$

Informally this rule states that the set of exceptions which may be raised by an application expression is the union of the sets which can be raised by evaluating the function value, the set of exceptions which may be raised by evaluating the argument and the set of exceptions which may be raised by the body of the function when applied to the given argument.

A further two rules may be added to the inference system to allow for expressions which always raise one of a set of exceptions. The first rule states that if the first expression in an application expression always raises one of a set of expressions, then the **any** type may be inferred for the whole application expression, since the argument expression will never be evaluated.

$$\frac{C \vdash e_1 \Rightarrow \{\mathbf{any}[\xi]\}}{C \vdash e_1\ e_2 \Rightarrow \{\mathbf{any}[\xi]\}} \tag{135}$$

This second rule states that if the argument expression always raises one of a set of exceptions then we may return the **any** type for the whole application expression, since the function will never be applied. However the exception annotation on this **any** type must incorporate all of the exceptions that may be raised by the evaluation of the function expression (note not the application of the function expression).

$$\frac{C \vdash e_1 \Rightarrow \overline{\tau}_1 \qquad C \vdash e_2 \Rightarrow \{\mathbf{any}[\xi]\} \qquad \tau[\xi''] \in \overline{\tau}_1 \text{ implies } \xi'' \subseteq \xi'}{C \vdash e_1\ e_2 \Rightarrow \{\mathbf{any}[\xi \cup \xi']\}} \tag{136}$$

Finally a rule for sub-typing exception annotations must be added, this rule allows a type τ_1 to be a sub-type of a type τ_2 if the undecorated types (ie without the exception annotations) are related by the subtype relation and the set of exceptions associated with τ_1 is a subset of, or equal to, the exception set associated with the type τ_2.

$$\frac{\tau_1 <: \tau_2 \qquad \xi \subseteq \xi'}{\tau_1[\xi] <: \tau_2[\xi']} \tag{137}$$

5.11.3.1 Exception Slack

The following section will describe an inference algorithm to compute delayed types equivalent to the sets of types which may be inferred for a given expression using the rules developed above. Before that, this section describes an advantage gained over a traditional strict typing scheme by using a delayed typing scheme.

It was mentioned in the introduction that the delayed typing scheme admits more programs which cannot go wrong than a type system based on Hindley-Milner. Recall that the programs for which it would be desirable to provide a type for but which a given type system cannot are known as the slack of the type system. One place where a delayed typing system includes more of the slack is expressions which will never be evaluated. For some such expressions a delayed typing scheme can give a delayed type which hides the type error. Because the value is never evaluated it is possible that the delayed type is thrown away and never reduced to a type error. A possible pragmatic use is in early software development or rapid prototyping where part of the program text does not yet pass type-checking. This can be blocked off by an always false conditional and allow testing to proceed on the part of the program which does pass type checking.

Once exceptions are introduced an expression may never be evaluated simply because it is within an exception handler, and the handler is never invoked because the related exceptions are never thrown in the appropriate context. Here is a worthwhile example: Suppose one has a function which performs some computation and will never raise a given exception E. But the programmer realises that it is possible that this function will change such that it may raise E.

let $work =$ **fun** $a \longrightarrow e$

If the programmer simply leaves this function as it is, it may eventually leak out a raised exception E which they did not mean to happen. A good way to avoid this is to wrap the value in a first class exception handler, for example:

let $handleE =$ **fun** $y \longrightarrow$ (**fun** $f \longrightarrow$ (**fun** $x \longrightarrow$ **try** $f\ x$ **with** $E \longrightarrow y$))

This function accepts a function f to apply, and an argument x, it also accepts a default answer y, which is returned in the case that the function application raises the exception E.

Now suppose our *work* function returns a list of items one way to wrap our function would be with:

let $workE = handleE\ []\ work$

This would work in most languages, however it is inherently bad programming style, if *work* is modified to possibly raise the exception E then it is likely that the program will have unexpected results since the arbitrary list value [] is returned with no justification.

Under delayed typing one could wrap the *work* function like this:

let $workE = handleE\ ()\ work$

The () expression would normally have the incorrect type however under delayed typing one of the types in the set inferred for the handle function would be:

$\forall('a\,'b).(() \rightarrow ('a \rightarrow 'b[]) \rightarrow 'a \rightarrow 'b)$

Here I have retained the empty exception annotation to highlight the fact that the functional argument must not raise the exception E. Recall that generally $\tau[]$ is equivalent to the type τ. There will of course be multiple types in the set which are similar but have different types for the first argument.

The key point is that with this definition, should the type of the *work* function change such that it may now raise the exception E, then the type system will reject the program. There is no type in the set which may be inferred for the $handleE$ function which can accept a unit type for the first argument and a function which may raise the exception E and returns a list for the second argument. At this point the programmer would be forced to reconsider what the correct answer should be whenever *work* raises the exception E.

5.12 Algorithm $\mathcal{W}^d$

In this section the algorithm for the inference of delayed types is developed. The typing rules allow for the inference of a set of types associated with an expression. The set of types appropriate for a given expression is commonly an infinite set and is hence awkward to display to the user. To avoid this a delayed type represents a set of types and it is delayed types which are inferred by the algorithm given in this section.

This section begins with a definition of the syntax of a delayed type. The algorithm for the basic lambda-calculus is then discussed.

5.12.1 Syntax of Delayed Types

In section 5.3.1 it was stated that a delayed type is a compact representation of a set of types. In this section that syntax is defined and in the following section an algorithm for inferring delayed types for expressions is given.

A delayed type is an expression waiting to be typed and is denoted as: $[[e]]$ The converse of a delayed type is a *concrete type*. A concrete type is a type which is not a delayed type.

In the algorithm that follows τ^c will be used to denote a concrete type.

The syntax for delayed types is an extension of the syntax for expressions to allow expressions to contain types. A delayed type can be thought of as a partially typed expression.

$$\begin{array}{rcll} e & := & c & \text{(constants)} \\ & | & \tau & \text{(types)} \\ & | & x & \text{(variable access)} \\ & | & \textbf{fun}\ x \rightarrow e & \text{(abstraction)} \\ & | & e_1\ e_2 & \text{(application)} \\ & | & \textbf{let}\ x\ =\ e_1\ \textbf{in}\ e_2 & \text{(let binding)} \end{array}$$

where types are augmented to allow the inclusion of a delayed type.

$$\tau \ := \ [[e]]$$

Notice that a concrete type may still contain a delayed type, however not at the top level. For example: $([[x]] \rightarrow [[e]])$ is a concrete type. A concrete type has a special status; all values which may be computed by the program have concrete type, in particular all top-level definitions will be given a concrete type. This is an important property since it is possible to mask a typing error within a delayed type, the typing error will be uncovered whenever the delayed type is 'reduced' to a concrete type. If this never happens, then it is because the expression associated with the delayed type is never evaluated.

Before giving the algorithm for delayed typing this section closes by briefly giving examples of a delayed type representing a set of types. The simplest example is that of the identity function, the algorithm below will infer the following type for the identity function:

$([[x]] \rightarrow [[x]])$

This represents the set of types, including type schemes, which are an arrow type in which the argument type is the same as the return type.

$\{(\tau_1 \rightarrow \tau_2) \mid \tau_1 = \tau_2\}$

Where sub-typing is included this set is increased to include all arrow types such that the argument type is a sub-type of the return type.

$\{(\tau_1 \rightarrow \tau_2) \mid \tau_1 <: \tau_2\}$

A delayed $[[e]]$ type will always be contained within an arrow type. This is because the expression within a delayed type should always contain at least one free variable, it is this free variable that is preventing the type from being reduced to a concrete type.

Here is a further delayed type $([[x]] \rightarrow [[x.lab]])$. This represents the set of all arrow types, whose argument is some record type containing the label *lab*, and whose return type is the type associated with the label *lab* in the argument record type.

5.12.2 The algorithm for delayed typing

Since the syntax for expressions is contained within the syntax for delayed types, our algorithm takes a delayed type as argument.

Recall that τ^c is a type which is not of the form $[[e]]$, that is, it is not a delayed type.

The algorithm also accepts as input a context C which maps identifiers to types. This is distinct from the contexts which were used in the typing inference rules which mapped identifiers to sets of types. Recall though that a delayed type is a concise representation of a set of concrete types. This is extended to typing contexts where a typing context used in this algorithm is a concise representation of the equivalent context for the typing rules.

As in the typing rules the context C_x represents the typing context obtained by removing all mappings of the identifier x from the typing context C. The algorithm then is given in Figure 5.4.

The cases for application expressions are worth highlighting. There are four kinds of application expressions which must be reduced by the algorithm, three of which are covered. The fourth kind is ill-typed application expressions which have no corresponding case in the algorithm and hence fail type-checking.

The first case is if the type of the function expression is delayed then the whole application expression is a delayed type. The type of the argument expression may or may not be delayed. In the second case the argument type is delayed and again the whole application expression is then given a delayed type. Note that in this case the function expression may not even have an arrow type, this will mean that ultimately the application expression is ill-typed, however if it is part of an expression that is never evaluated the type-system need not reject it and hence a delayed type is returned. The third case involves a concrete function type and a concrete argument type. In this case the argument type is substituted into the body of the function type to give the resulting type which may or may not be delayed. Finally if both types are concrete but the type of the function expression is not an arrow type then the application expression is ill-typed and no type is assigned to the expression.

5.12.3 Notes

The case for the typing of a constant expression uses the function $\mathcal{CON}$ as used in the typing rule 113. However this function returns a set of types, but the algorithm should return a single delayed type. For the purposes of this algorithm however the $\mathcal{CON}$ function will return a delayed type which is equivalent to the set of types associated with the constant. In practice the set of types for a constant is commonly a singleton set such as $\{\mathbf{int}\}$, returning an equivalent delayed type is therefore trivial.

It may seem strange that there is no rule which reduces an arbitrary arrow

```
$\mathcal{W}^d(C,e) = \tau$
  If $e$   is $\mathbf{int}$
           then $\tau = \mathbf{int}$
  If $e$   is $\mathbf{bool}$
           then $\tau = \mathbf{bool}$
  If $e$   is $'a$
           then $\tau = {}'a$
  If $e$   is $\forall('a).\tau'$
           then $\tau = \forall('a).\mathcal{W}^d(C,\tau')$
  If $e$   is $c$
           then $\tau = \mathcal{CON}(c)$
  If $e$   is $x$
           then $\tau = C(x)$
  If $e$   is $\mathbf{fun}\ x \to e_1$
           then $\tau = ([[x]] \ \to\ \mathcal{W}^d(C_x[x \mapsto [[x]]], e_1))$
  If $e$   is $e_1\ e_2$
           and $\mathcal{W}^d(C,e_1) = [[e']]$      $\mathcal{W}^d(C,e_2) = \tau_2$
           then $\tau = [[e'\ \tau_2]]$
  If $e$   is $e_1\ e_2$
           and $\mathcal{W}^d(C,e_1) = \tau_1$      $\mathcal{W}^d(C,e_2) = [[e']]$
           then $\tau = [[\tau_1\ e']]$
  If $e$   is $e_1\ e_2$
           and $\mathcal{W}^d(C,e_1) = ([[x]] \ \to\ [[e']])$      $\mathcal{W}^d(C,e_2) = \tau_2^c$
           then $\tau = \mathcal{W}^d(C_x[x \mapsto \tau_2^c], e')$
  If $e$   is $\mathbf{let}\ x\ =\ e_1\ \mathbf{in}\ e_2$
           and $\mathcal{W}^d(C,e_1) = [[e_1']]$      $\mathcal{W}^d(C_x[x \mapsto [[x]]], e_2) = \tau_2$
           then $\tau = [[\mathbf{let}\ x\ =\ e_1'\ \mathbf{in}\ \tau_2]]$
  If $e$   is $\mathbf{let}\ x\ =\ e_1\ \mathbf{in}\ e_2$
           and $\mathcal{W}^d(C,e_1) = \tau_1^c$
           then $\tau = \mathcal{W}^d(C_x[x \mapsto \tau_1^c], e_2)$
```

Figure 5.4: Algorithm for delayed typing

type. There is no rule which might look something like:

If e is $(\tau_1 \rightarrow \tau_2)$
  and $\mathcal{W}^d(C, \tau_1) = \tau_3$ $\mathcal{W}^d(C, \tau_2) = \tau_4$
  then $\tau = (\tau_3 \rightarrow \tau_4)$

This is intentional. Without type constraints all function types are of the form $([[x]] \rightarrow [[e]])$. The next section will introduce how the algorithm handles type constraints, as part of this, arbitrary arrow types must be considered.

The square brackets indicating a delayed type are only considered necessary at the outermost level or the outermost level within a concrete type. This means that types containing superfluous square brackets are considered equal, for example: $[[e_1\ e_2]]$ is equal to the type $[[[[e_1]]\ [[e_2]]]]$ and similarly $[[\mathbf{let}\, x = e_1 \mathbf{\ in\ } e_2]]$ is equivalent to $[[\mathbf{let}\, x = [[e_1]] \mathbf{\ in\ } [[e_2]]]]$

5.12.4 Additions to the Typing Algorithm

In this section additions to the type system are incorporated into the algorithm for delayed typing.

5.12.4.1 Type Constraints

To allow for the typing of expressions using type constraints the extra cases may be added to the inference algorithm shown in Figure 5.5. However they require the definition of an auxiliary function I^d. This matches two types and, if the given two types are compatible, returns a substitution mapping the bound identifiers in the first type to the appropriate component-types of the second type. There will only be bound identifiers in the first type if it contains delayed types. The algorithm for I^d is shown later in Figure 5.6, before this the complications which arise from the addition of type constraints are discussed.

The addition of type constraints complicates the typing algorithm for the remaining expression kinds. This is because those cases dealing with such expressions cannot now expect that each type is a *fully delayed* type. A fully delayed type is one in which all bound identifiers have been delayed and all

```
$\mathcal{W}^d(C,e) = \tau$
  If $e$   is $e : \tau^c$
           and $\mathcal{W}^d(C,e) = [[e']]$
           then $\tau = [[e' : \tau^c]]$
  If $e$   is $e : \tau^c$
           and $\mathcal{W}^d(C,e) = \tau_1$      $I^d(\tau_1,\tau^c) = S$
           then $\tau = \mathcal{W}^d(C, S(\tau_1))$
  If $e$   is $\textbf{let}\, x : \tau^c = e_1 \textbf{ in } e_2$
           and $\mathcal{W}^d(C,e_1) = [[e']]$      $\tau_2 = \mathcal{W}^d(C[x \mapsto [[e']]], e_2)$
           then $\tau = [[\textbf{let}\, x : \tau^c = e' \textbf{ in } \tau_2]]$
  If $e$   is $\textbf{let}\, x : \tau^c = e_1 \textbf{ in } e_2$
           and $\mathcal{W}^d(C,e_1) = \tau_1$      $I^d(\tau_1,\tau^c) = S$
           then $\tau = \mathcal{W}^d(C[x \mapsto S(\tau_1)], e_2)$
  If $e$   is $\textbf{fun}\, x : \tau^c \to e$
           and $\tau_1 = \mathcal{W}^d(C_x[x \mapsto \tau^c], e)$
           then $\tau = (\tau^c \rightarrow \tau_1)$
```

Figure 5.5: Algorithm cases for delayed typing of type constraints

typing decisions based on their use delayed. These complications are briefly discussed before the algorithm for the I^d function is detailed.

Previously in the absence of type constraints, all types could be considered to be fully delayed. In particular this meant that a function type was always of the form:

$([[x]] \rightarrow [[e]])$

Application of such a type was simple because the argument type was substituted into the delayed result type for the type $[[x]]$. With the addition of typing constraints it cannot be assumed that a function type is so simple, it may have the more general form:

$(\tau_1 \rightarrow \tau_2)$

where τ_1 may contain any number of delayed types, and in particular, as was done for the identifier x in the simplest case, there may be identifiers which must be given a type corresponding in some way to the type of the argument to which the function is applied. For example an application expression $e_1\ e_2$ may be typed in the context where e_1 is given type

$(([[x]] \rightarrow [[x]]) \rightarrow [[x\ \mathbf{int}]])$

and where e_2 is given type

$(\mathbf{int} \rightarrow \mathbf{int})$

This application should be typed correctly but there is currently no case which will apply to this situation. A new case, which is a more general form of the previously defined case for application, is required. The two cases which deal with a delayed type for the function or argument expression are retained. It is still desirable to return a delayed type in the case that either of the sub-expressions are given a delayed type. Therefore in this new case both sub-expressions are assumed to have been given a concrete type, the function expression must have an arrow type.

If e is $e_1\ e_2$
and $\mathcal{W}^d(C,e_1) = (\tau_2 \rightarrow \tau')$ $\quad \mathcal{W}^d(C,e_2) = \tau_2^c$ $\quad S = I^d(C,\tau_2,\tau_2^c)$
then $\tau = \mathcal{W}^d(C,S(\tau'))$

In order to perform the application this case uses the I^d function. The algorithm for this function is given in Figure 5.6.

Note that this algorithm uses union over substitutions, denoted by $\cup_{merge}$. This fails if both substitutions map the same identifier to different types.

$$\begin{array}{llcl}
I^d(C\,,[[x]] & ,\tau^c) & = & [x \mapsto \tau^c](\text{ if } x \notin Dom(C)) \\
I^d(C\,,\mathbf{int} & ,\mathbf{int}) & = & [] \\
I^d(C\,,\mathbf{bool} & ,\mathbf{bool}) & = & [] \\
I^d(C\,,(\tau_1 \rightarrow \tau_2) & ,(\tau_3 \rightarrow \tau_4)) & = & I^d(C,\tau_1,\tau_3) \cup_{merge} I^d(C,\tau_2,\tau_4) \\
I^d(C\,,\forall('a).\tau_1 & ,\forall('b).\tau_2) & = & I^d(C,\tau_1,['b \mapsto 'a]\tau_2)
\end{array}$$

Figure 5.6: Algorithm for the instantiation of types

An instantiation may fail due to the non-concreteness of the left-hand side. In fact the only delayed type on the left hand side which can be instantiated is the single identifier $[[x]]$.

For example the following instantiation will fail.

$I^d([[\mathbf{let}\, x \;=\; e_1 \;\mathbf{in}\; e_2]], \tau^c)$

This is because there is not enough information to instantiate the bound identifier x. It is unknown how the type of e_1 should be constrained such that it is a concrete type. However this does not necessarily indicate a failure in the typing of an expression, merely that the typing of the expression should be delayed within a delayed type. Consider the following function application in the environment where $+$ is a predefined function of type ($\mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int}$):

$(\mathbf{fun}\, x \longrightarrow \mathbf{let}\, y : \mathbf{int} \;=\; 1 + x \;\mathbf{in}\; y)\; 1$

For the algorithm to infer a type for this application expression, it must first act recursively on the abstraction expression. This will in turn recurse through to the let binding containing the type constraint. At this point the algorithm types the initialising expression and is assigned the delayed type $[[(\mathbf{int} \;\rightarrow\; \mathbf{int})\; x]]$

When the algorithm attempts to instantiate the type **int** to this type it does not know where to start. Hence the instantiation fails. However this failure is acceptable because it is a delayed type. The function may be given type $[[\mathbf{fun}\, x \longrightarrow [[\mathbf{let}\, y : \mathbf{int} \;=\; (\mathbf{int} \;\rightarrow\; \mathbf{int})\; x \;\mathbf{in}\; [[y]]]]]]$

The typing of the entire application expression may now proceed as the argument expression (the integer constant 1) is given type **int** which is then applied to the abstraction type. The initialisation type of the delayed **let** type can now be given type **int** and type constraint applied. The whole application expression is finally given type **int**.

5.12.4.2 Record Fields

The addition of record fields does not complicate the algorithm which can work recursively over the fields of a record creation expression. The interesting decision is when to return a delayed type and when to return a concrete type. A record type is distinct from a function type in that a function type is always concrete even if the constituent types are delayed. A record type however is only concrete if each of the labelled types are themselves concrete.

The justification for this is the masking of type errors. Consider the following application $\textbf{fun}\ x \rightarrow ((\textbf{fun}\ r \rightarrow r.lab_1)\ \{lab_1 = 1\ ; lab_2 = x\ 1\ ;\})$

The type of the initialising expression of lab_2 is delayed, however if the record type was considered concrete then the algorithm could reduce the application of the inner abstraction to the record expression and the delayed type of lab_2 would be lost. If eventually the outer abstraction is applied, then the initialising expression for lab_2 is still evaluated (as this is a strict language), even though it is not used. Hence if the value to which the outer abstraction is applied is not a function which can accept an integer argument then the typing algorithm would not detect the type error. To prevent this a record type is considered to be a delayed type unless all of the constituent label types are themselves concrete.

The algorithm therefore makes use of a function $\overline{fr}$ which can take a delayed record type $[[\{lab_n = \tau_n\ ;^+\}]]$ and if all of the τ_n types are concrete then a concrete record type $\{lab_n : \tau_n\ ;^+\}$ is returned, otherwise the original delayed record type is returned. Note the difference between the field initialisations $lab = \tau\ ;$ in a delayed record type and the field declarations $lab : \tau^c\ ;$ in a concrete record type. The first line of the algorithm operates over delayed record types and the second line operates over concrete record types – all the label types within are concrete but those concrete types may contain delayed types which may be reducible.

The algorithm for field rows; $\mathcal{W}^{df}(C, fields) = fields'$ must operate both on field rows as expressions (and part of a delayed type) and also on field types. In both cases the algorithm is straightforward and calls the main $\mathcal{W}^d$ function recursively on the types of all the labels. The $\mathcal{W}^{df}$ function is shown in figure 5.8

5.12.4.3 Recursion

Because inference of recursive definitions is always assisted by the programmer in the form of a provided type constraint, the extensions to the algorithm are simplistic. The point to note here is that although the type constraint is a

```
If $e$   is $\{fields\}$
         and $fields' = \mathcal{W}^{df}(C, fields)$
         then $\tau = \overline{fr}([[\{fields'\}]])$
If $e$   is $\{fieldecs\}$
         and $fieldecs' = \mathcal{W}^{df}(C, fieldecs)$
         then $\tau = \{fieldecs'\}$
If $e$   is $e.lab$
         and $\{lab : \tau_1 \,; \langle fieldec \rangle\} = \mathcal{W}^d(C, e)$
         then $\tau = \tau_1$
If $e$   is $e.lab$
         and $[[e']] = \mathcal{W}^d(C, e)$
         then $\tau = [[e'.lab]]$
```

Figure 5.7: The additional cases to support record typing.

concrete type, the initialising expression may still be a delayed type since the whole **let** expression may be nested within another expression. Therefore the first additional case allows for this situation and returns a delayed type of the whole recursive binding. The second additional case performs the reduction of the recursive definition.

```
If $fields$   is $label = e \,; \langle fields \rangle$
              and $\mathcal{W}^d(C, e) = \tau_1$      $\langle \mathcal{W}^{df}(C, fields) = fields' \rangle$
              then $fields_1 = (label = \tau_1 \,; \langle fields' \rangle)$
If $fields$   is $label : \tau^c \,; \langle fieldecs \rangle$
              and $\mathcal{W}^d(C, \tau^c) = \tau_1^c$      $\langle \mathcal{W}^{df}(C, fieldecs) = fieldecs' \rangle$
              then $fields_1 = (label = \tau_1^c \,; \langle fieldecs' \rangle)$
```

Figure 5.8: The algorithm for the $\mathcal{W}^{df}$ function.

If e is **let rec** $x : \tau^c \;=\; e_1$ **in** e_2
and $\mathcal{W}^d(C_x[x \mapsto \tau^c], e_1) = [[e']]$ $\quad \mathcal{W}^d(C_x[x \mapsto [[x]]], e_2) = \tau_2$
then $\tau = [[$**let rec** $x : \tau^c \;=\; e'$ **in** $\tau_2]]$

If e is **let rec** $x : \tau^c \;=\; e_1$ **in** e_2
and $\mathcal{W}^d(C_x[x \mapsto \tau^c], e_1) = \tau_1$ $\quad I^d(\tau_1, \tau^c) = S$
then $\tau = \mathcal{W}^d(C_x[x \mapsto S(\tau_1)], e_2)$

5.13 Properties

This section describes the properties of the delayed typing system and the associated inference algorithm detailed in sections 5.4 and 5.12.

5.13.1 Subsumes Hindley-Milner Typing

The type system should have the property that it includes all of the Hindley-Milner type system. This means that if a term is typable under the Hindley-Milner type system [62] then it should also be typable under delayed typing. The Hindley-Milner type system is defined with the set of inference rules given in Figure 5.9

In this section a proof of this property is detailed.

To avoid confusion separate typing relations are used for delayed typing and Hindley-Milner typing. The term: $C \vdash e \;\Rightarrow_{HM} \tau$ will mean that under the the Hindley-Milner type system, the expression e may be given the type τ in the typing context C.

In contrast the term: $C \vdash e \;\Rightarrow_D \overline{\tau}$ will mean that the expression e under the delayed typing scheme given the typing context C has the set of types $\overline{\tau}$. Note that the typing contexts used by the sets of rules are different for Hindley-Milner and delayed typing.

Recall from section 5.12.2 that the algorithm uses typing contexts of the kind used in the Hindley-Milner inference rules. That is, identifiers are mapped to single types. Two typing contexts C_1 and C_2 in Hindley-Milner and delayed typing styles respectively, are considered equivalent if for every mapping $x \mapsto \tau$

$$\frac{\mathcal{CON}(c) = \tau}{C \vdash c \Rightarrow_{HM} \tau} \tag{138}$$

$$\frac{C \vdash e \Rightarrow_{HM} \sigma \qquad \sigma' = inst(\sigma)}{C \vdash e \Rightarrow_{HM} \sigma'} \tag{139}$$

$$\frac{C \vdash e \Rightarrow_{HM} \sigma \qquad \alpha \text{ not free in } C}{C \vdash e \Rightarrow_{HM} \forall\alpha.\sigma} \tag{140}$$

$$\frac{C(x) = \sigma}{C \vdash x \Rightarrow_{HM} \sigma} \tag{141}$$

$$\frac{C \vdash e_1 \Rightarrow_{HM} (\tau_1 \to \tau) \qquad C \vdash e_2 \Rightarrow_{HM} \tau_1}{C \vdash (e_1\ e_2) \Rightarrow_{HM} \tau} \tag{142}$$

$$\frac{C_x[x \mapsto \tau_1] \vdash e \Rightarrow_{HM} \tau}{C \vdash \mathbf{fun}\, x \to e \Rightarrow_{HM} (\tau_1 \to \tau)} \tag{143}$$

$$\frac{C \vdash e_1 \Rightarrow_{HM} \sigma \qquad C_x[x \mapsto \sigma] \vdash e_2 \Rightarrow_{HM} \tau}{C \vdash \mathbf{let}\, x = e_1\ \mathbf{in}\ e_2 \Rightarrow_{HM} \tau} \tag{144}$$

Figure 5.9: Typing rules for the Hindley-Milner type system

in C_1 there is a mapping in C_2 such that $x \mapsto \overline{\tau} \wedge \tau \in \overline{\tau}$. The same variable will be used across both styles of typing contexts and as such to mean that the two are equivalent.

Using this convention the desired property of delayed typing can be stated more formally as:

Lemma 5.13.1. *if* $C \vdash e \Rightarrow_{HM} \tau$ *then* $C \vdash e \Rightarrow_D \overline{\tau}$
and that $\tau \in \overline{\tau}$.

Equivalently we have that, given the assumption above then $C \vdash e \Rightarrow_D \{\tau\}$ since the subsumption rule (111) may be used.

The proof is done by induction on the size of the deduction which is a witness to the assumption.

Proof. Case Taut: $C \vdash e \Rightarrow_{HM} \sigma$ By the rule for tautology in Hindley-Milner it must be the case that $C(x) = \sigma$ therefore by the assumptions we can assume that $C(x) = \overline{\tau}$ and that $\sigma \in \overline{\tau}$. Therefore rules 111 and 114 can be used to deduce $C \vdash e \Rightarrow_D \{\sigma\}$

Case Inst:

$$\frac{C \vdash e \Rightarrow_{HM} \sigma \qquad \sigma' = inst(\sigma)}{C \vdash e \Rightarrow_{HM} \sigma'}$$

by induction then we can assume that $C \vdash e \Rightarrow_D \{\sigma\}$. Hence by the use of the delayed typing instantiation rule 110 we have $C \vdash e \Rightarrow_D \{\sigma\} \cup \{\sigma'\}$

Case Gen:

$$\frac{C \vdash e \Rightarrow_{HM} \sigma \qquad \alpha \text{ not free in } C}{C \vdash e \Rightarrow_{HM} \forall \alpha.\sigma}$$

by induction we have $C \vdash e \Rightarrow_D \overline{\tau}$ and $\sigma \in \overline{\tau}$ hence by rule 112 we can deduce $C \vdash e \Rightarrow_D \overline{\tau} \cup \{\forall('a).\sigma\}$

Case Comb:

$$\frac{C \vdash e_1 \Rightarrow_{HM} (\tau_1 \rightarrow \tau) \qquad C \vdash e_2 \Rightarrow_{HM} \tau_1}{C \vdash (e_1\ e_2) \Rightarrow_{HM} \tau}$$

By induction we know that we must have $C \vdash e_1 \Rightarrow_D \{(\tau_1 \rightarrow \tau_2)\}$ and $C \vdash e_2 \Rightarrow_D \{\tau_1\}$ hence by rule 116 we must have $C \vdash e_1\ e_2 \Rightarrow_D \{\tau\}$ as required.

Case Abs:

$$\frac{C_x[x \mapsto \tau_1] \vdash e \Rightarrow_{HM} \tau}{C \vdash \textbf{fun}\ x \rightarrow e \Rightarrow_{HM} (\tau_1 \rightarrow \tau)}$$

By the induction hypothesis we must have that $C_x[x \mapsto \{\tau_1\}] \vdash e \Rightarrow_D \{\tau\}$ so from rule 115 there must be a $\overline{\tau}$ such that $\tau \in \overline{\tau}$ and $C \vdash \textbf{fun}\ x \rightarrow e \Rightarrow_D \overline{\tau}$

Case Let:

$$\frac{C \vdash e_1 \Rightarrow_{HM} \sigma \qquad C_x[x \mapsto \sigma] \vdash e_2 \Rightarrow_{HM} \tau}{C \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Rightarrow_{HM} \tau}$$

By the induction hypothesis we must have that $C \vdash e_1 \Rightarrow_D \{\sigma\}$ and also by the induction hypothesis we have $C_x[x \mapsto \{\sigma\}] \vdash e_2 \Rightarrow_D \{\tau\}$ and so by rule 117 we have $C \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Rightarrow_D \{\tau\}$ □

5.13.2 Non-Ambiguity

The type system should be unambiguous. This means that if $C \vdash e \Rightarrow \overline{\tau}_1$ and $C \vdash e \Rightarrow \overline{\tau}_2$ then there exists a set of types $\overline{\tau}$ such that $C \vdash e \Rightarrow \overline{\tau}$ and both $\overline{\tau}_1$ and $\overline{\tau}_2$ are subsets of, or equal to, $\overline{\tau}$.

A small lemma named the inclusion lemma is required first in order to show the main theorem of non-ambiguity.

Lemma 5.13.2. *The inclusion lemma states that if $C_x[x \mapsto \overline{\tau}_1] \vdash e \Rightarrow \overline{\tau}$ and $\overline{\tau}_2 \supseteq \overline{\tau}_1$ then $C_x[x \mapsto \overline{\tau}_2] \vdash e \Rightarrow \overline{\tau}'$ and $\overline{\tau}' \supseteq \overline{\tau}$.*

The proof of this lemma is simple and rests on the fact that either the expression e does not contain the identifier x, in which case the same deduction used for $C_x[x \mapsto \overline{\tau}_1] \vdash e \Rightarrow \overline{\tau}$ can be used to deduce $C_x[x \mapsto \overline{\tau}_2] \vdash e \Rightarrow \overline{\tau}$. If the expression e does contain the identifier x then any deduction will involve at least one step using rule 114 to deduce that $C_x[x \mapsto \overline{\tau}_1] \vdash x \Rightarrow \overline{\tau}$ and this can be replaced by a two step reduction using both the subsumption rule (rule 111) and the aforementioned rule 114 to deduce $C_x[x \mapsto \overline{\tau}_2] \vdash x \Rightarrow \overline{\tau}$ since $\overline{\tau}_1 \subseteq \overline{\tau}_2$.

5.13.2.1 The main non-ambiguity theorem

The main result of non-ambiguity for the delayed typing system can be achieved with induction on the size of the expression e.

Case $e = c$ Then $C \vdash e \Rightarrow \mathcal{CON}(c)$ one can therefore infer a subset of the set of types, $\overline{\tau}$ in $\mathcal{CON}(c)$ but for any two such type sets they have the common super set of types $\overline{\tau}$.

Case $e = x$ For any $\overline{\tau}$ such that $C \vdash e \Rightarrow \overline{\tau}$ it must be the case that we have $\overline{\tau} \subseteq C(x)$ Therefore any two such sets of types must both be a subset of, or equal to, the set $C(x)$.

Case $e = \textbf{let}\ x\ =\ e_1\ \textbf{in}\ e_2$ If $C \vdash e \Rightarrow \overline{\tau}'$ then by the rule for let typing (rule 117) $C \vdash e_1 \Rightarrow \overline{\tau}_1'$ and $C_x[x \mapsto \overline{\tau}_1'] \vdash e_2 \Rightarrow \overline{\tau}'$ and similarly if $C \vdash e \Rightarrow \overline{\tau}''$ then $C \vdash e_1 \Rightarrow \overline{\tau}_1''$ and also $C_x[x \mapsto \overline{\tau}_1''] \vdash e_2 \Rightarrow \overline{\tau}''$

By the induction hypothesis there must exist $\overline{\tau}_1$ such that $C \vdash e_1 \Rightarrow \overline{\tau}_1$ and $\overline{\tau}_1 \supseteq \overline{\tau}_1'$ and $\overline{\tau}_1 \supseteq \overline{\tau}_1''$. Hence by the inclusion lemma it must be the case that $C_x[x \mapsto \overline{\tau}_1] \vdash e_2 \Rightarrow \overline{\tau}_2'$ and that $\overline{\tau}_2' \supseteq \overline{\tau}'$ and $C_x[x \mapsto \overline{\tau}_1] \vdash e_2 \Rightarrow \overline{\tau}_2''$ and that $\overline{\tau}_2'' \supseteq \overline{\tau}''$. Finally by the induction hypothesis there must exist a $\overline{\tau}_2$ such that $\overline{\tau}_2 \supseteq \overline{\tau}_2'$ and $\overline{\tau}_2 \supseteq \overline{\tau}_2''$ and $C_x[x \mapsto \overline{\tau}_1] \vdash e_2 \Rightarrow \overline{\tau}_2$ and by the typing rule for let expressions (rule 117) $C \vdash e \Rightarrow \overline{\tau}_2$ as required.

Case: $e = (e_1\ e_2)$

If $C \vdash e \Rightarrow \overline{\tau}'$ then by the rule for application typing (rule 116) $C \vdash e_1 \Rightarrow \overline{\tau}_1'$ and $C \vdash e_2 \Rightarrow \overline{\tau}_2'$ and the set $\{\tau \mid (\tau_2\ \rightarrow\ \tau) \in \overline{\tau}_1' \wedge \tau_2 \in \overline{\tau}_2'\}$ is equal to, or a super set of, $\overline{\tau}'$.

Similarly if $C \vdash e \Rightarrow \overline{\tau}''$ then by the rule for application typing (rule 116) $C \vdash e_1 \Rightarrow \overline{\tau}_1''$ and $C \vdash e_2 \Rightarrow \overline{\tau}_2''$ and the set $\{\tau \mid (\tau_2\ \rightarrow\ \tau) \in \overline{\tau}_1'' \wedge \tau_2 \in \overline{\tau}_2''\}$ is equal to, or a super set of, $\overline{\tau}''$.

Via the induction hypothesis it must be the case that $C \vdash e_1 \Rightarrow \overline{\tau}_1$ and $\overline{\tau}_1 \supseteq \overline{\tau}_1'$ and $\overline{\tau}_1 \supseteq \overline{\tau}_1''$ and similarly we have $C \vdash e_2 \Rightarrow \overline{\tau}_2$ and $\overline{\tau}_2 \supseteq \overline{\tau}_2'$ and $\overline{\tau}_2 \supseteq \overline{\tau}_2''$

Due to these type set relations we must have the set $\overline{\tau} = \{\tau \mid (\tau_2\ \rightarrow\ \tau) \in \overline{\tau}_1 \wedge \tau_2 \in \overline{\tau}_2\}$ and this set must be a super set or equal to both the sets $\overline{\tau}'$ and $\overline{\tau}''$. Finally then the typing rule for application can be applied and we have that $C \vdash e \Rightarrow \overline{\tau}$

5.13.3 Type Safety

An important property of the delayed typing scheme is one of type safety. This means that if the delayed typing scheme allows a type to be inferred for a given expression then the program cannot "go wrong". It must first be defined what it means for a program to "go wrong" and indeed to do this a suitable evaluation function must first be defined.

The definition for both and the basis for the proof of type safety are taken from [63]. An outline of the proof is given in this section.

The evaluation function proceeds from left to right by re-writing. The evaluation function, as in [63], requires a partial function $\delta : \mathit{Constant} * \mathit{Value} \rightharpoonup \mathit{Value}$ which interprets the application of functional constants to values. The δ used must have the following $\delta - \mathit{typability}$ property: if $\mathcal{CON}(c) = (\tau_1 \rightarrow \tau_2)$ and $v : \tau_1$ then $\delta(c, v)$ is defined and $\vdash \delta(c, v) \Rightarrow \{\tau_2\}$

For a program to "go wrong" the evaluation must get stuck when attempting to reduce an expression. This occurs when an expression is reduced to $(c\ v)$, which represents a constant applied to a value, for which $\delta(c, v)$ is not defined. In particular if we have $(e_1\ e_2)$ and the expression e_1 is neither further reducible nor of the function form $(\mathbf{fun}\, x \rightarrow e)$ then the expression is said to be stuck and hence to have "gone wrong".

An evaluation step is written as $e_1 \Longrightarrow e_2$. The closure of this operation is written as: $e \longmapsto v$ and for a program that does not succeed we have: $e \longmapsto \mathit{WRONG}$.

The property we wish to prove for delayed typing then becomes: if
$\vdash e \Rightarrow \{\tau\}$ then
$e \not\longmapsto \mathit{WRONG}$

The proof of this rests on the notion of subject reduction. This states that reductions preserve the type of an expression. Formally, if $\vdash e_1 \Rightarrow \{\tau\}$ and $e_1 \Longrightarrow e_2$ then $\vdash e_2 \Rightarrow \{\tau\}$

Subject reduction is not enough on its own to ensure type safety, it must also be shown that an expression whose reduction has become stuck because of some type error, for example $(1\ 1)$, cannot be typed. If this were not the case, then an expression which is well-typed can still be reduced to a stuck expression without violating subject reduction.

For the simple lambda-calculus there is only one kind of expression which is stuck. That is an expression $(c\ v)$ for which $\delta(c, v)$ is undefined. Such expressions cannot be given a type otherwise the $\delta - typability$ condition is violated.

5.13.3.1 Subject Reduction

The proof of subject reduction proceeds via induction on the size of the expression being reduced. The interesting case is given below and rests upon the replacement lemma. The replacement lemma allows the replacement of one sub-expression of a typable expression with another sub-expression of the same type without altering the type of the whole expression. Using $e[e_1]$ to mean an expression e with a 'hole' in it which is filled by the sub-expression e_1 then the replacement lemma can be formally stated as follows.

Lemma 5.13.3. *If* $C \vdash e[e_1] \Rightarrow \{\tau\}$ *and* $C_1 \vdash e_1 \Rightarrow \{\tau_1\}$ *and* $C_1 \vdash e_2 \Rightarrow \{\tau_1\}$ *then* $C \vdash e[e_2] \Rightarrow \{\tau\}$

The interesting case for subject reduction then is that of application:
Case: $(\mathbf{fun}\ x \rightarrow e_1)\ v \Longrightarrow e_1[x \mapsto v]$
From the assumption $C \vdash (\mathbf{fun}\ x \rightarrow e_1)\ v \Rightarrow \{\tau\}$ and the rule for abstraction (115) we must have that: $C \vdash v \Rightarrow \{\tau_1\}$ and $C \vdash \mathbf{fun}\ x \rightarrow e_1 \Rightarrow \{(\tau_1 \rightarrow \tau_2)\}$ for at least one τ_1 and τ_2. From the rule for abstraction it must be the case that $C_x[x \mapsto \{\tau_1\}] \vdash e_1 \Rightarrow \{\tau_2\}$ By the replacement lemma we have that $C \vdash e_1[x \mapsto v] \Rightarrow \{\tau_2\}$

5.13.4 Exception Safety

In this section the type system extended to include the inference of exception annotations as discussed in section 5.11 is considered. An important property of any system which infers sets of exceptions which may be raised is one of *exception safety*. This means that if an expression e_1, when evaluated may raise an exception E_1, then any type inferred for e_1 is annotated with a set of exceptions which includes E_1.

For the delayed type system described here this property can be formally written as:

Lemma 5.13.4. *If $C \vdash e \Rightarrow \overline{\tau}$ and $e \longmapsto$ **raise** E then $\tau[\xi] \in \overline{\tau}$ implies that $E \in \xi$.*

Recall that τ is a synonym for $\tau[]$ so if $C \vdash e \Rightarrow \overline{\tau}$ and $\tau \in \overline{\tau}$ then this requires that $e \not\longmapsto$ **raise** E for any E.

A small fact about deductions is used, this states:

Lemma 5.13.5. *If $\vdash e \Rightarrow \{\tau[\xi]\}$ then any deduction for the type of e which involves this deduction, must deduce a set $\overline{\tau}$ in which every member of $\overline{\tau}$ is annotated with at least the set $[\xi]$.*

The main property of exception safety can be proven with the use of subject reduction. Informally, subject reduction states that; if the type system allows an expression e_1 to be typed with a given set of exceptions as an annotation and e_1 can be reduced to e_2 then the type system allows the inference of the same set of exceptions for expression e_2. In particular if the type system allowed a smaller set of exceptions to be inferred for the expression e_1 than was possible for e_2 then the property of subject reduction would not hold.

Formally then the property of subject reduction is given by :

Lemma 5.13.6. *If $\vdash e_1 \Rightarrow \overline{\tau}_1$ and $e_1 \Longrightarrow e_2$ then $\vdash e_2 \Rightarrow \overline{\tau}_2$ and $\tau[\xi] \in \overline{\tau}_1$ implies $\tau[\xi] \in \overline{\tau}_2$*

Since the only rule which applies to **raise** expressions is rule 131 we have: $\vdash$ **raise** $E \Rightarrow \{$**any**$[E]\}$ and by the property 5.13.5 there is no type $\tau[\xi]$ which may be inferred as part of a set for the **raise** expression such that $E \notin \xi$. Hence for all **raise** expressions any type which may be inferred for it must have the raised exception as part of the exception annotation. By subject reduction if an expression e_1 may be reduced to a **raise** expression then all types in any set inferred for the expression e_1 must contain the raised exception in the associated annotation set. Since it would be impossible to infer a set which did not contain the raised exception for the **raise** expression. Therefore subject reduction implies exception safety.

The interesting case for subject reduction for exception safety is now shown. Case: $(\textbf{try raise } E \textbf{ with } E \rightarrow e_2) \Longrightarrow e_2$ By the assumption we have that $\vdash \textbf{try raise } E \textbf{ with } E \rightarrow e_2 \Rightarrow \overline{\tau}$ for some non-empty set. By rule 131 we have $\vdash \textbf{raise } E \Rightarrow \{\textbf{any}[E]\}$ and lemma 5.13.5 we have that when using rule 132 the set $\overline{\tau}_3$ is empty since none of the types in the set for the first expression do not contain the exception E. This means that the set inferred for the whole **try** expression is equal to the set inferred for the expression e_2. Since the **try** expression reduces to e_2 subject reduction is preserved. □

In general subject reduction holds for exception safety because for each form of expression the exceptions which it may raise depend upon the exceptions which may be raised by the sub-expressions. Each rule is deliberately tailored not to allow any exceptions to escape without annotation.

5.14 Comparisons

In this section some related type-systems and type-system extensions aimed at solving similar problems to that which the delayed typing system of Nitro is aimed at are detailed and compared to the delayed typing system.

5.14.1 Hindley-Milner

The Hindley-Milner type system[64] is the basis upon which much work on type-systems is founded. This provides polymorphic typing of programs in the simply-typed lambda-calculus. Not only the basis for much research on type-systems, the Hindley-Milner type system is also the essence of many type systems in practical use, including but not limited to, the type systems of SML, OCaml, Clean [65] and Haskell. Recall from Section 5.13.1 the set of inference rules for the Hindley-Milner type system given in Figure 5.9

In section 5.13.1 it was proved that the delayed typing system can provide a type for all programs which the Hindley-Milner typing system can, therefore the typing system types as many programs. Since it was also shown in section 5.13.3 that the delayed typing system does not allow the typing of any 'wrong' programs the delayed typing system can be said to include the Hindley-Milner type system. This is a good starting point as it means we have not produced a type system which is less general than or provides less guarantees than a type system which is in widespread use.

It is simple to give an example of a program which is typable under the delayed-typing system but not in Hindley-Milner. The simplest example involves the polymorphic use of an argument type. In delayed typing a type can be given to the following function:

$\mathbf{fun}\ f \rightarrow (f\ 1, f\ \mathit{true})$

The delayed type assigned to it is:

$([[f]]\ \rightarrow\ [[(f\ \mathbf{int}, f\ \mathbf{bool})]])$

A traditional Hindley-Milner type system will however assign no type to the above function.

In this sense the delayed-typing system can be said to have included some of the *slack* of the Hindley-Milner typing system. The slack of any type system is the set of programs which are not-typable but which it is desirable to type. Often this set corresponds to those programs which although they are typable are not 'wrong', in the sense that the program will always either fail to terminate or terminate with a value of the correct type.

As noted in the section 5.11.3.1 another kind of program some of which are typable under delayed typing but not under Hindley-Milner are those which contain expressions which are guaranteed not to be evaluated. Although this class of programs are generally uninteresting there are, as noted, opportunities to use this ability to increase the maintainability of some software.

5.14.2 Rank 2 - Rank N Polymorphism

The 'rank' of a type system refers to the depth at which universal quantifiers may appear in a contra-variant position. Because the original Hindley-Milner type system is a rank 1 system no polymorphism occurs in the argument of a functional argument. This was demonstrated in the example function of the previous section. The functional argument f cannot be applied to both an integer and a boolean because it cannot be polymorphic in a rank 1 system. In a rank 2 system however one would be able to type such a function.

Rank-2 polymorphism allows a function to accept a polymorphic argument. With Rank-1 polymorphism all $\forall$ quantifiers come at the outer scope of a type. For example one may have
$\forall(\alpha).(\alpha \rightarrow \alpha)$

but one cannot have the type
$\forall(\alpha).(\forall(\beta).(\beta \rightarrow \beta) \rightarrow \alpha)$

The delayed typing system is quite capable of giving a type to such functions. In fact a delayed typing system is an arbitrary-rank type system, that is polymorphism can be nested however deep in the contra-variant position as required. It is known that pure type inference becomes difficult or intractable for rank ≥ 2. However the delayed typing scheme described in this thesis requires type constraints on recursive functions and hence does not have pure type inference. It therefore does not automatically follow that the delayed-typing system is intractable. Some work on making arbitrary-rank polymorphic type systems tractable includes [44].

There remain some programs outside of the scope of an arbitrary-rank polymorphic type system which can be typed by a delayed type system. One such class has already been mentioned, those which include un-typable expressions which are guaranteed not to be evaluated at run-time.

Additionally giving a Hindley-Milner type system higher-rank polymorphism does not address the problem of sub-typing. Recall that sub-typing was a requirement of the new delayed typing system for Nitro because of the need to encode foreign type information in abstraction-level code.

5.14.3 Abstract Interpretation

Abstract Interpretation[66] is a means of static analysis which consists of executing the program but in an imprecise manner. Each point in the program is given not a specific value but a set of possible values which it may take on a given execution of the program. Often this set is an interval of values. In this way a single abstract interpretation of the program can approximate all possible runs of the program on all the inputs (or a sub-set of the possible inputs in which the analyser is interested).

A delayed typing scheme is similar to abstract interpretation, whereby the set of types represents the set of possible values. In fact delayed typing may be seen as a specific instance of abstract interpretation where the abstractions are fixed at the level of the type. The programmer may alter the precision of the abstraction by using more refined types since the delayed typing scheme admits sub-typing. While delayed typing loses some of the generality of abstract interpretation it gains in simplicity for the programmer. No extra definitions are required on the part of the programmer, and the same abstraction is used on every program therefore a programmer new to a specific software project is immediately familiar with the static analysis used.

It is possible that abstract interpretation could be integrated into a delayed typing scheme and that such a marriage would yield a powerful design. It may be that the advantages of conformity across software projects and specific analyses can be combined with the advantage of greater control over the granularity of the specific analyses. This may be a promising area for subsequent work.

5.14.4 Soft-Typing

The main idea behind soft-typing as described in [5] is to combine the expressivity of dynamic typing with the feedback given to the programmer before execution of the program offered by static typing. The main idea is to use a static type system, but where a program is ill-typed, rather than produce an error and refuse to compile the program, a warning is emitted and a dynamic run-time check is inserted. This can be seen as an extension to the common method for typing arrays in statically typed programming languages. Generally it is not possible to ensure statically that the index of an array access is within the bounds of the array being accessed. Therefore the compiler will allow all such accesses but insert a dynamic run-time check that the index is within the bounds of the array before executing the actual access. Soft-typing extends this method to the more general question of whether a value is used appropriately.

This method marries two advantages of the static and dynamic typing strategies; the ability to detect errors at compile time coupled with the ability to run all correct programs. However the boundary still exists in practice, since the extra correct programs which are now allowed, but would be failed by a static type system, are dynamically typed. This means that no extra programs are statically approved.

Introducing automatic run-time checks has the disadvantage that the compiler may need to introduce a private run-time representation of values in order to determine their types at run-time. This was something that Nitro sought to avoid.

5.14.5 Dependent Types

A dependent type is a type which depends upon a value. Dependent types[67] can ensure more properties of a program than can a delayed type. For example in a language providing dependent types one may write a *zip* function over two lists and ensure that it is only called with two lists of the same length. This is not possible using delayed types for arbitrary lengths of lists. It may be possible using index types to provide something similar but in general delayed types cannot provide the same guarantees which a dependent type is capable of.

Dependent types have been incorporated into traditional Hindley-Milner type systems, such as with dependent ML (see [68]) and there is no reason why dependent types could not be incorporated into a delayed typing system.

5.14.6 Existential Types

The relationship of abstract data types to existential types was first described by Mitchell and Plotkin [69]. Existential types have been used in Nitro to assist in the typing of bare array accesses. Their use allows a polymorphic structure to hold heterogeneous items about which local constraints can be retained. In the case of Nitro the 'local constraints' were limited to equality. This allowed the Nitro programmer to define an abstract array data type which held both the array itself and its length. The existential types allow one to ensure that the length argument given is indeed the length of the bare array, but also that this length did not have to be any particular length. This was necessary as it allows one to equate the types of two arrays of differing lengths, provided their lengths are stored together with the corresponding array. Without this ability it would, for example, be impossible to choose between two arrays unless both arrays were created with the same index variable.

5.14.7 Constraint Solving

A constraint solving type system, such as HM(X)[70] first described by Odersky, Sulzmann and Wehr, has much in common with a delayed typing system. A delayed type can be seen as a representation of the constraints upon the type. Instead of a further typing language used to describe the constraints, the constraints are implicit in the form of the delayed expression. For example the type
$([[f]] \rightarrow [[f\ \mathbf{int}]])$

constrains the argument type to be a function which can accept values of type **int** (or a sub-type of **int**).

5.14.8 Intersection Types

An intersection type is a type belonging to two types. An intersection type is usually written as $\tau_1 \wedge \tau_2$ and means a type which is both of type τ_1 and τ_2. This only makes sense if intersection types are used in the presence of a sub-typing relation. Without a sub-typing relation then we have simply that $\tau = \tau_1 \wedge \tau_2$ if τ is an instantiation of both τ_1 and τ_2.

In the presence of sub-types an intersection type becomes more powerful.

In fact the power of expressivity of intersection types is such that type inference is known to be undecidable. However this does not mean that the type system is unusable.

In Nitro a delayed conditional type is equivalent to an intersection type. Consider the simple if function

$\textbf{fun}\ b \rightarrow (\textbf{fun}\ x \rightarrow (\textbf{fun}\ y \rightarrow \textbf{if}\ b\ \textbf{then}\ x\ \textbf{else}\ y))$

This could be given the intersection type

$(\textbf{bool} \rightarrow ('a \rightarrow ('b \rightarrow 'a \wedge 'b)))$

in delayed typing we would give the type:

$([[b]] \rightarrow ([[x]] \rightarrow ([[y]] \rightarrow [[\textbf{if}\ b\ \textbf{then}\ x\ \textbf{else}\ y]])))$

Given a suitable typing for conditionals, in which the end result is a common super-type of the types of both branches, then both the intersection and the delayed types for the given function can be applied to the same set of types. Intersection types were studied by Benjamin Pierce for his PhD thesis [71, 72].

5.14.9 Sized Types

Sized types can have slightly different meanings, but generally sized types relate static information about the magnitude of values in a program, as in [73]. The size information typically relates somehow to the number of objects within a container type such as the elements in a list or an array. The size information is often in the form of a relation, rather than reasoning about the absolute size of data structures the size relative to each other is calculated. For example the *append* function which joins two lists together. The size type will specify that the returned list is larger than both of the input lists, or perhaps even that it is exactly equal to the size of the first plus the size of the second.

Size types can also be used to ensure the termination of recursive functions which traverse structurally inductive data types, by ensuring that the size of the argument given to all the recursive calls is smaller than the size of the 'current' argument.

In a similar way to dependent types, sized types offer more than a delayed type, but are not mutually exclusive to delayed types. That is one can imagine a delayed type system incorporating some form of sized types. Sized types do offer more than the index types used by Nitro's foreign data interface to allow the encoding of array length information. The Nitro solution is however more lightweight.

5.14.10 Uniqueness Types

A uniqueness type guarantees that an object has at most one reference to it. When this is guaranteed then destructive update may be used upon it to improve the efficiency of functional code without relinquishing the important property of referential transparency. The language Clean [65] uses uniqueness types to provide input/output actions in a lazy functional programming language [74].

As with dependent types and sized types, uniqueness types can in theory be incorporated into a delayed typing system.

5.14.11 Haskell Type Classes

Type classes as used in Haskell (see [75, 76, 77]) offer a method to allow ad-hoc polymorphism into functional programs. This system is not orthogonal to the use of delayed types, in fact since type classes can benefit from a sub-typing relation, adding a type class mechanism to a delayed typing system would be complimentary.

5.15 Incorporation Into Nitro

This section describes how the delayed typing scheme was incorporated into the Nitro programming language.

5.15.1 Syntax

There was relatively little concrete syntax in the way of expressions which were added to Nitro from that which already existed from the introduction of the foreign data constructs as described in Chapter 4.

Before the inference of exception annotations Nitro did not have any exception raising or handling facilities. This was because it is inherently unsafe to allow the propagation of an exception from the abstraction level code to the calling procedure. Now that exception annotations may be accurately inferred to indicate which exceptions may be raised by the abstraction one can either ensure that no such exceptions will leave the abstraction level or at least be aware of the possibility. This second option allows a system to prevent the number of such escaping exceptions to be maintained, or even constrained. Furthermore when one does cause a problem there is hope that the offending portion of code can be tracked down more easily than if another abstraction-level programming language had been used.

The syntax for raising and catching exceptions added to Nitro is as follows:

$$\begin{array}{lll} e & := & \textbf{raise } E \\ & \mid & \textbf{try } e_1 \textbf{ with } E \rightarrow e_2 \end{array}$$

5.15.2 Tagged Union Data Types

Tagged union data types were not mentioned in the discussion of delayed types. However tagged types play a central role in the foreign data interface facilities of Nitro.

A tagged type can be represented in a similar way to record types and indeed tagged types work very well with the delayed typing system. There is a natural sub-typing relation over tagged union types which is explained in the following section.

5.15.3 Sub-typing

As with record types there is a sub-typing relation between tagged union data types. However the sub-typing relation is the converse of that used for record types. Recall that a record type is a sub-type of another record type if it contains all of the same record fields. A tagged type is a sub-type of another tagged type if it contains no constructors which are not also present in the second type.

The rational for this is that a function expecting an argument which is a tagged type containing the constructors $con_1, ..., con_n$ will perform well on a value which was created with any one of the given constructors (and the appropriate argument type). Such a value will have a tagged type which may contain less constructors that the set $con_1, ..., con_n$. If this function is applied to a value with a tagged type containing a constructor not in that set, then the function may fail, in particular if the function matches over the constructor then the pattern match will fail. So a function with type $((con_1\ \tau_1|....con_2\ \tau_2) \rightarrow \tau)$ may be applied to any with a tagged type containing a sub-set of those constructors $(con_1\ \tau_1|....con_2\ \tau_2)$, but not to a tagged type containing any constructors out with that set.

Conversely a function excepting a record type which contains the labels $lab_1, ..., lab_n$ can accept a value which was created with more labels defined but not less. If it is applied to a value created with less labels defined, then one of the undefined labels may be accessed by the function, causing an error.

As with record fields though, the parameter types of the constructors must be sub-types. Here are a couple of examples before the sub-typing relation for tagged types is more concretely defined.

$(label1\ \textbf{of int}) <: (label1\ \textbf{of int} \mid label2\ \textbf{of bool})$

$(label1\ \textbf{of int} \mid label2\ \textbf{of}\ \forall('a).('a \rightarrow 'a)) <:$
$\quad (label1\ \textbf{of int} \mid label2\ \textbf{of}\ \forall('a).(\textbf{bool} \rightarrow \textbf{bool}))$

Here are the additional two rules to allow sub-typing for tagged union types under delayed typing. They use the $\mathcal{P}$ function which, given a tagged union type τ and a constructor label lab returns the parameter type of the constructor lab within the type τ. If τ is not a tagged union type or does not contain the constructor lab then the function $\mathcal{P}$ fails.

$$\frac{\tau_1 <: \mathcal{P}(\tau, label) \qquad \langle constrs <: \tau \rangle}{(label\ \textbf{of}\ \tau_1 \langle \mid constrs \rangle) <: \tau} \tag{145}$$

5.15.4 Bare Arrays

This section defines the additional typing rules which must be added to the delayed typing inference rules to allow the typing of the foreign data bare arrays. The key points here are that we cannot assign the same index variable to more than one value (though we can of course alias an identifier with an index type) and that an array can only be assigned one index variable type hence it cannot be accessed with the wrong index variable.

$$\frac{C \vdash e_1 \Rightarrow \{\mathbf{int}\} \qquad C_i[i \mapsto \{\mathbf{index}.(a)\}] \vdash e_2 \Rightarrow \overline{\tau} \qquad a \text{ not free in } C}{C \vdash \mathbf{let\ index}\ i\ = e_1\ \mathbf{in}\ e_2 \Rightarrow \overline{\tau}} \tag{146}$$

Since arrays are mutable the type of the cells within an array can only have one type, not a set of types. This is the same as a mutable record field.

$$\frac{C \vdash e_1 \Rightarrow \{\tau\} \qquad C \vdash e_2 \Rightarrow \{\mathbf{index}.(i)\}}{C \vdash \mathbf{array}\ e_1\ e_2 \Rightarrow \{\tau\ \mathbf{array}.(i)\}} \tag{147}$$

$$\frac{C \vdash e_1 \Rightarrow \overline{\tau} \qquad C \vdash e_2 \Rightarrow \{\mathbf{index}.(i)\} \qquad C \vdash e_3 \Rightarrow \{\mathbf{int}\}}{C \vdash e_1.(e_2).[e_3] \Rightarrow \{\tau \mid \tau\ \mathbf{array}.(i) \in \overline{\tau}\}} \tag{148}$$

$$\frac{C \vdash e_1 \Rightarrow \{\tau\ \mathbf{array}.(i)\} \qquad C \vdash e_2 \Rightarrow \{\mathbf{index}.(i)\} \qquad C \vdash e_3 \Rightarrow \{\mathbf{int}\} \qquad C \vdash e_4 \Rightarrow \{\tau\}}{C \vdash e_1.(e_2).[e_3] \leftarrow e_4 \Rightarrow \{()\}} \tag{149}$$

Chapter 6

Regions

This chapter details the region memory management scheme used in Nitro. Regions allow the programmer to control the lifetimes of objects created by the Nitro code while retaining a type-safety guarantee. This ability is essential for abstraction-level code which may create values the lifetimes of which depend on higher-level code.

This chapter begins with a background look at different memory management schemes, most of them involving a compromise between the two extremes of automatic garbage collection done at runtime and fully manual memory management where the whole issue of memory management is left up to the programmer. This is completed by an introduction to the general regions memory management scheme. Nitro regions are then introduced by detailing the language constructs which are available to the programmer for manipulating data storage in regions of memory. The typing of such region constructs within the delayed typing scheme of the previous chapter is described at the user-level. In section 6.3 a formal semantics for the delayed typing of region constructs is shown. The chapter concludes with a discussion of the success of using a region memory management scheme within an abstraction-level functional programming language such as Nitro.

6.1 Background

When designing a language a choice must be made on how to manage memory within programs written with that language. There are several choices that can be split into two major forms. Explicit memory management in which the programmer has full control over memory operations, and automatic memory management in which the programmer leaves the task to the compiler and runtime services. Several schemes lie somewhere in between these two possibilities and usually involve a compromise between the advantages of each.

6.1.1 Explicit Memory Management

If explicit memory management is chosen then the programmer generally has three operations available. The programmer must be able to request a new block of memory and to delete or relinquish control of any block obtained in this way. Finally the programmer is given the ability to reuse a block of memory, this may simply be a side-effect of the language allowing arbitrary update.

There are several advantages to explicit memory management. The three most notable are control, efficiency and predictability. By allowing programmers control over the management of memory they can rely on some properties of a program that would not be guaranteed by an automatic system of memory management. This can be necessary particularly when manipulating memory is a central part of the program being developed rather than just a necessity required to solve the real problem. Such control can also lead to an increase in efficiency, though efficiency is listed as a separate advantage because the explicit operations for maintaining the memory of a program are usually simple operations compared with the dynamic bookkeeping that must be done by an automatic runtime service, however the measured cost of per-allocated cell may be less with a dynamic garbage collector (see below in Section 6.1.2). In addition to control and efficiency, where programmers explicitly allocate and free portions of memory they have the predictability of such operations. Furthermore not only are such operations predictable in their behaviour but they are often evident in the source code of the program. This means that the programmer cannot only predict a stall in some operation due to memory management routines but is also more likely to notice the possibility of such a stall in advance of a test procedure uncovering it.

There are also several disadvantages to explicit memory management. The two major disadvantages are the inconvenience it forces upon the programmer and the lack of a safety guarantee. Because programmers have full control over the reuse and lifetimes of the memory blocks which they request, the only thing preventing them from making a mistake is themselves. The previous paragraph mentions that programmers may use properties of the program which they as programmers can know but which cannot be expressed to the compiler or runtime system. However this means that the programmer can also make use of a property of the program which is not true and therefore access a memory location with an incorrect type.

The inconvenience to programmers is perhaps the least attractive drawback of explicit memory management. For every program the programmers must spend a large fraction of their time solving an issue that is not central to the main problem presented by the program which they are trying to implement. In addition the source code of the program is sprinkled with calls to invoke memory operations. This can help highlight bugs and deficiencies in the memory management part of the program but it can also obscure the implementation details of the real problem. In other words using explicit memory management can make the program harder to read.

6.1.2 Implicit Memory Management

Automatic memory management usually takes the form of a runtime garbage collector. When the garbage collector is invoked it scans the variables in the program that are still live. This involves searching down the stack of the program and the machine's program registers. The garbage collector assumes that all such variables may be used again and hence can be considered live. It must determine which of these variables constitute pointers into the heap memory of the program. All portions of the memory in the heap which are not reachable from the live variables in the program are considered to be *garbage* that is safe to delete or reuse. There are several different strategies or algorithm for the garbage collector to use, many are reviewed and discussed in [78].

The advantages of automatic runtime garbage collection mirror the disadvantages of explicit memory management. The convenience to the programmer can greatly improve productivity. The language is now able to make a guarantee of safety; since the programmer need not be provided control over the deletion and reuse of memory, the language can ensure that these operations are only performed when it is safe to do so. This is of course not the only requirement for type safe programming but it does make it possible.

The disadvantages are clear – the loss of control over the reuse and deletion of memory can mean that memory which will never again be accessed by the program is retained by the program simply because the garbage collector cannot determine that the memory will indeed never be accessed again. A programmer employing an explicit memory management scheme may use properties of the program which a dynamic garbage collector must spend time computing while the program is running. The loss of efficiency in speed is generally noted as a disadvantage of using a runtime garbage collector. The actual cost is not always as great as perceived, for example in [79] Appel notes that as the size of the memory is increased the cost of deallocating a cell of memory can be made to tend towards zero. In [80] Zorn measures the cost of conservative garbage collection by using explicit and implicit garbage collection with the same programs. His conclusions indicate that the cost in extra CPU cycles of garbage collection is low and questionable. However he also concludes that programs using automatic collection will have an increased use of memory. For environments where there are small caches and memory spaces this can degrade performance.

Controlling when the garbage collector makes a pass to determine the garbage in the heap is one of many similar problems that are presented to a language offering runtime garbage collection. A lot of research has been directed at these problems and many solutions exist although many compromise some of the advantages of automatic runtime garbage collection. For example a language may provide a 'collect-garbage-now' operation that the programmer may invoke, generally during an inactive period of the program – such as between the levels of a game – to attempt to avoid the collector running during a more critical phase of the program. However any programmer using this has lost some of the convenience and readability advantages that were gained by using automatic garbage collection.

Furthermore the lack of explicit memory management operations makes programs in which the central problem is the manipulation of memory difficult or perhaps impossible to write. Other such low-level tasks become awkward such as the implementation of a marshalling routine to allow the interoperation between two high-level languages. As was discussed in chapter 4 such implementation tasks require low-level access to the structure of data but even with this ability they can be awkward to develop if the programmer cannot explicitly create or delete memory. Because of this compiler writers must implement many of the low-level routines provided by the language in C or some other low-level language rather than the programming language which the compiler is for.

6.1.3 Compromise Solutions

So far the two extremes of memory management have been considered, but there are many other schemes that fall somewhere between the two. Before moving on in section 6.1.4 to discuss region based memory management, included here is a brief survey of some techniques that offer a compromise between explicit and manual memory management.

6.1.3.1 Checked Explicit Memory Management

One alternative is checked explicit memory management (for examples [81]: the programmer must explicitly invoke the memory management operations but the compiler can check these to make sure that memory is not reused or deleted too soon. By employing this a programming language can provide some of the advantages of explicit memory management while capturing a major advantage of a runtime garbage collector, that is programs can be ensured to be type safe. However this scheme does not gain the advantage of convenience of automatic garbage collection.

This general scheme can be further split into two different categories: the compiler may check the memory management operations at compile time or insert runtime code to apply the check dynamically. To enable compile-time checking there are often restrictions on when these commands may be called. This offers programmers most of the benefits of efficiency and predictability. There is still a greater control over memory than with an automatic scheme but not quite as much as with unchecked explicit memory management. For example a programmer cannot use properties of a program that cannot be presented to the compiler.

Checking each memory management command at runtime offers the orthogonal advantages of retaining control but losing much of the efficiency and predictability. In fact in order to speed up such calls often the runtime checking – for example checking a delete operation is safe – is postponed until it is actually required – in this case, when the memory is reused. This means that the pause required to make the runtime check is unpredictable both in its duration and in when it is performed. As a result this scheme is little used as a primary memory management scheme, however it has been employed as a means of improving legacy code. In such cases a combination of altering the semantics of pointer and array accesses to speed up the runtime checks and static analysis to remove such checks where they are not required is used. The Safe-C compiler uses a technique described in [22] to provide efficient checking of pointer and array accesses. The CCured project [19, 21] analyses the input C program to determine the smallest number of runtime checks that must be inserted to make the program memory safe – where memory safe means that the program will stop rather than overrun buffers or write over data that is still in use. In addition the analysis performed can highlight some bugs that would otherwise require tests to discover.

6.1.3.2 Implicit Compile-Time Memory Management

Automatic compile-time memory management hopes to automatically insert in the program memory management commands, for example see [82]. This means that a program written for use with an automatic runtime garbage collector is translated into an explicitly managed program.

In theory this can offer the best of both worlds. From automatic garbage collection the programmer retains safety and convenience but still does not sacrifice efficiency of the translated program. Predictability can still be retained though only if the programmer is aware of the inference used for automatically inserting the memory management commands. Control is nearly lost, though again with some knowledge of the underlying scheme the programmer is often able re-write their program in order to steer the inference algorithm in some way. In practice such systems have proven very difficult to implement. Generally the inference of where in the program to place the memory re-use command is very conservative and as a result the program retains unused data for too long. An example of an inference system for an ML-like programming language with promising early results is described in [83] with the remaining problems due to polymorphism and mutable reference cells.

6.1.3.3 Compile-Time Garbage Collection

The above section introduced the notion of automatically inferring the memory management commands so that the programmer need not insert them, but also a runtime garbage collector was not required. The notion of *compile-time garbage collection* is to automatically infer when it is safe to insert memory re-use commands but still combine this with a runtime garbage collector for example [84]. In [85], Mazur, Ross, Janssens and Bruynooghe describe the implementation and design of one such system for the Mercury [86] programming language.

Where compile-time garbage collection is used static analysis techniques are employed to reduce the runtime cost of the garbage collector. Because memory is reused there is less garbage to collect, also when and what areas of memory need to be collected can be inferred at compile-time. To the programmer the scheme is essentially an automatic runtime garbage collected scheme but performance is often improved. Once again the scheme can be difficult to implement.

6.1.3.4 Other Possibilities

Finally it is possible to offer some finer grain compromises by combining parts of the schemes above. For example combining automatic compile-time memory management and checked explicit memory management can result in what is for the most part automatic compile-time memory management but which allows the insertion of constraints to make sure the inference has detected key points at which to delete or reuse memory. This provides some measure of control and predictability at the cost of some of the convenience. Another example is allowing the programmers to designate which values should be automatically reclaimed by a garbage collector and which they wish to relinquish manually perhaps using checked explicit memory management commands.

In [87] a linear type system for a first-order functional language is presented which allows in-place update by ensuring that all references are only used once (hence the linear quality of the type system). The main innovation is a resource type $\diamond$ which is bound to the space used by a constructor, such as the list constructor *cons*. When a value is deconstructed via a pattern match its resource may be re-used thus allowing in-place update. Since the resource may only be used once it is guaranteed not to be needed anywhere else. In [88] the authors present an updated type system which is less restrictive. This allows some sharing of data by allowing more than one access to the references of the resource type $\diamond$. Provided that all but the last use is non-destructive this is shown not to interfere with the semantics of the translated program. This work is in the context of the MRG or Mobile Resource Guarantees project[89], which aims to provide machine checkable proofs about the resources consumed by a given program. The resource in question is often maximum memory usage such that a program can be determined to be safe to run on a device with a limited amount of memory.

For the most part this section has talked about 'the language' offering a certain scheme and for most of the time this meant 'the implementation' of a language. The compromise schemes are often not part of the language but are attached by a specific implementation often as an extra program analyser or library. The possible reason for this is that any one language does not wish to tie itself down to one particular scheme lest some other superior scheme emerges. It may also be simply because memory management innovators are concerned with the deployment of a particular memory management scheme rather than the nuances of language design. In any case most languages stipulate only explicit or automatic memory management and the precise scheme used is left up to the implementors and, sometimes, even the programmers themselves, for example the Boehm-Demers-Weiser[45, 90] conservative garbage collector provides C programmers with automatic runtime garbage collection. For some schemes language support is desirable or even necessary. In these cases we often see a dialect of an existing language emerge.

6.1.4 Region Memory Management

So far this section has discussed memory management schemes by distinguishing them based on the amount of work done by the programmer, the compiler and any runtime services. Between the two extremes of entrusting everything up to the programmer and providing completely automatic memory management by the runtime system, several compromise positions have been identified. This subsection will detail a method, *regions*, that can be employed to strike a balance between several of the compromise solutions. Regions have been used for several decades one of the first uses within an explicit memory management system was described by Ross in [91] and [92] gives a good survey of the use of regions for memory management splitting those listed into three kinds; those that ensure safety statically, those that ensure safety dynamically and those which are unsafe.

A region is a portion of memory in which a set of objects may be stored. Regions are often expandable as more objects are stored within the region. Alternatively it is possible to group together regions. Within a region-based memory management scheme objects created within a program may be stored within a region. Regions may be created and deleted dynamically at runtime. When a region is deleted all of the objects stored within that region are also deleted. Generally an object may be stored in at most one region though there is often a sub-region relation in which all objects stored within a sub-region may be seen as being stored within another region that is guaranteed to outlive the first.

Regions may be programmed by hand, explicitly creating and deleting regions. The deletion operations are not checked by the compiler. When so used region programming is explicit memory management and cannot guarantee safety. In this sense the regions are used to structure the explicit memory control in the hope that mistakes are less likely and can also improve performance as all values allocated within a region can be deallocated with one delete region operation. Furthermore although there are no compile-time guarantees as to the safety of the program it is possible to add dynamic runtime checks whenever the "delete region" operation is called. Work on explicit region memory management and allowing language support for such constructs includes [93, 92].

6.1.4.1 Stack of Regions

In [94], Tofte and Talpin first presented the stack of regions memory management scheme as a means of allowing static checking and automatic inference for region based memory management. Regions are maintained using a stack. When a region is created it is placed upon the region stack. A region cannot be deallocated unless it is at the top of the region stack. Therefore regions have the property that any region created is destroyed before any of the regions that were still alive at the time of its creation. When an object is created it may be stored within any of the regions that are currently live. The object is deleted whenever the region in which it is contained is itself deleted. An object may only live in one region.

Definition 1 (Region scheme). *For the remainder of this thesis when referring to a regions scheme this will mean a region scheme in which there is a stack of regions which must be deleted according to the LIFO discipline of stack management.*

Within a region scheme there are three main operations. A `create_region` operation to create a new region on the top of the region stack and a handle to be used to refer to the new region. An `allocate_region` operation allocates space for a new object within a region. Finally a `delete_region` operation deletes the region and all of the values that have since been allocated within that region. This operation may require as a parameter the region to delete and check that it is the top of the region stack, or it may implicitly delete the top of the stack.

As before these operations may be invoked explicitly by the programmer. When this is done either there is no safety guarantee or safety is ensured by static type checking, dynamic runtime checks or a combination of both. Additionally such region operations may be inferred by the compiler. An algorithm for such an inference is given in [95].

Regions in this form then can be used to provide a range of solutions to memory management which find a compromise between the control and performance of explicit memory management and the safety and convenience of automatic runtime garbage collection. Explicit region operations that are checked by the compiler offer some of the control and performance of explicit memory management, while also providing the safety of automatic garbage collection at the expense of much of the convenience. If the checking is done statically then some of the control is lost as some programs that are safe to run are rejected because the type system is not powerful enough to express it. If the checks are dynamic much of the control can be retained but at the cost of a loss in performance, though this loss may not be significant.

6.1.4.2 Region Inference

Inferring region operations automatically means that the safety and convenience of dynamic garbage collection may be retained. However, to a large extent the control that explicit memory management provides is lost. There is also a loss of some of the performance afforded by explicit regions as the inference does not always infer the optimal regions into which to place objects. Some of this can be offset by the programmer, with some knowledge of the inference algorithm, by rewriting their program to be more friendly to the inference engine. A better approach is to simply allow explicit annotations but still infer the region operations where the programmer has not given them. In either case some of the convenience afforded by entirely automatic dynamic garbage collection is lost, but at the significant benefit of a gain in performance, control and predictability.

Regions as used in the ML-Kit[96], are automatically inferred by the compiler. This was done because the authors were compiling an existing language that did not contain region primitives. Used in this way regions are a form of automatic compile-time memory management which it has already been noted appears to offer the best of both worlds however in practice the user must have some knowledge of the region system.

6.1.4.3 Cyclone Regions

The Cyclone language has extended[97] this approach by adding into the language explicit region annotations for the programmer to assist the inference algorithm by giving manual hints as to where might be good places to insert region operations. In addition this has the advantage of helping to maintain region code that has been fine tuned. For example a programmer who has manipulated the way in which a piece of code has been written in order to get the best region performance from the compiled code can ensure that future updates to the code do not invalidate properties which the fine tuning relied upon.

In addition several memory management techniques have been incorporated into Cyclone such that programmers have a choice of approaches for each object in their programs. For example programs may store objects in a special Heap region which is never deleted and may be optionally garbage collected at runtime. There is also a special Unique region. Objects stored in this region have only one reference to them and therefore can be deleted as soon as the pointer leaves the active scope. Although this can burden the programmer with choice it can also offer them the control and performance comparable with explicit memory management when required.

The Nitro language uses an approach similar to Cyclone. A region memory management scheme is used, region operations are given explicitly by the programmer and checked to be safe by the compiler. A separate heap that may be optionally garbage collected is also offered. The remainder of this chapter concentrates on detailing the typing of region operations within Nitro and in particular how this interacts with the delayed typing scheme detailed in Chapter 5.

6.2 Nitro Regions

The example Nitro code in the previous chapters takes no notice of memory management. The code was written with the assumption that all values are allocated on the heap and automatically collected by a runtime garbage collector. This can be done using a conservative garbage collector without modifying or requiring an internal representation of values; recall that this was an important property for Nitro since otherwise foreign values must be separated from internal Nitro values.

The use of such a garbage collector is frequently inappropriate, for example when writing a marshalling routine to allow the exchange of data between two other high-level languages. Most of the values created should be registered with the garbage collector of the target language, some intermediate values used during the translation should be deleted immediately. This section introduces the region primitives provided by Nitro and details the typing involved at the level appropriate for a user of the type system. In section 6.3 the syntax and typing of the region constructs are formalised.

6.2.1 The Type of Regions

Regions are given unique region names. The type of a region is written **region.**(v) where v is the name of the region. A region variable is a region name that may be instantiated. We write **region.**('r) to denote a region variable. Only a region variable can be instantiated; a region name cannot.

6.2.2 The let region construct

Regions can be created with the **let region** construct, this has the form **let region** r **in** e_1. In the expression e_1, the name r has type **region.**(v) where v is a unique region name. The syntax of region types is similar to that of index types but there is a subtle difference, distinct region names may be unified. Whereas one index type may never be used in place of another, some region types may be interchangeable. This is because there is a natural sub-typing relation between regions based on lifetimes.

During the evaluation of the expression **let region** r **in** e_1, once the expression e_1 has been evaluated the region r is removed, and all of the objects which were created within that region are relinquished. The type system must make sure that no such object is later accessed.

6.2.3 The at construct

The **at** construct allows the creation of a new object inside a particular region. It has the form e_1 **at** r where r is a program variable which must have type **region.**(v) for some region name v. The object is created within the region v. When that region is removed the object is automatically relinquished. Some expressions create no new data. In this case the **at** construct has no effect at runtime, however it does affect the typing of the expression. In this case the **at** annotation becomes a constraint which specialises the type of the expression.

The expression must have a type with an **at** annotation specifying a region which lives longer than the region specified by the **at** annotation of the expression. This is best described by way of an example:

```
let example =
  let region r1 in
  let region r2 in
  let p = (1, 2) at r1 in
  let q = p at r2 in
    q
```

During the typing of this expression the identifier `p` has type `(`**`int`**`, `**`int`**`)` **`at`** `r1`. The expression `p` **`at`** `r2` is allowed since the region `r1` outlives the region `r2`. The identifier `q` has type `(`**`int`**`, `**`int`**`)` **`at`** `r2` even though it refers to the same actual value as `p`, because the annotation **`at`** `r2` given to the expression `p` – which does not produce any data – acts as a region constraint.

6.2.4 The Uses Region Type

The Uses region type specifies which regions may be accessed by the computation of the value associated with the type. It is written as $\tau.\{R\}$. Here R is a set of regions which may be accessed during the evaulation of the expression to which the type system has inferred the type τ.

6.2.5 The no Type

The type checker must invalidate the types of values stored in regions which go out of scope. In addition it must invalidate the type of functions which may access regions which go out of scope. However it is not desirable to reject all code for which a value escapes the scope of the region it is allocated within provided that value is never accessed outside of that scope. Similarly functions which access regions out of scope are allowable if they are never applied.

To allow for this the **no** type is introduced. This is the opposite of the **any** type which was introduced in section 5.11. Recall that the **any** type could be unified with any other type. It was used to indicate that instead of a value being returned an exception would definitely be raised, hence it could be unified with any other type. When unified with any other type, the resulting type is the other type.

By contrast the **no** type, when unified with any other type, always returns the **no** type. This is because it is not safe to use a value of type **no**. Since the unification of any other type together with the **no** type returns the **no** type, a value of **no** type cannot be used with any operation. So a **no** type can be part of a tuple or record type, it may also be the type of a returned value. However a value with type **no** cannot be manipulated or inspected. Generally a **no** type is the result of a compound value such as a tuple value going out of scope, so the pointer which represents the tuple is safe to be passed around, but cannot be dereferenced.

6.2.6 The Heap Region

The heap region is a special region that is never deallocated. Any expression that does not have an associated `at` construct is equivalent to an expression with an `at` construct with the heap region as the given region. This means that a program which ignores regions altogether can still be accepted. The heap region itself may be optionally garbage collected using the Boehm-Demers-Weiser conservative garbage collector. For some programming tasks the combination of automatic garbage collection with specific regions for some values works well. For others one or other approach is preferable. The Cyclone developers have experimented with several safe memory management idioms and the use of two or more together within a single program. A report on these experiments can found in [81].

6.3 Syntax

This section begins the formalisation of the region typing within the domain of delayed typing. In this section the syntax for the region expressions and types are provided and in section 6.4 the static semantics of region programs are formalised.

The additional syntax for region programming in Nitro is given in Figure 6.1. The additional types are given in Figure 6.2.

$$\begin{array}{lll} \mathit{expr} & := & \textbf{let region } r \textbf{ in } \mathit{expr} \\ & | & \mathit{expr} \textbf{ at } r \end{array}$$

Figure 6.1: The additional syntax of expression for region programming.

$$\begin{array}{lll} \tau & := & \textbf{region}.(v) \\ & | & \tau \textbf{ at region}.(\mathit{region}) \\ & | & \tau.\{\mathit{region}\} \\ & | & \textbf{no} \\ \mathit{region} & := & v \\ & | & {}'r \\ & | & H \end{array}$$

Figure 6.2: The additional types for region programming

Although the region types are types, the programmer may consider them to be more like annotations. To allow for this the following set of rules define equivalence over types involving region types. This allows distinct types to be treated as equal types in much the same way as distinct type schemes are treated as equal if they differ only in the names of the bound type variables.

A type without an enclosing region type may be considered equivalent to one specifying the H. So that τ is equivalent to τ **at** $H.\{H\}$.

Nested annotations are combined so that: $(\tau$ **at** $r_1)$ **at** r_2 is equivalent to τ **at** r where r is equal to the shortest living region between r_1 and r_2 or the longest living region if the type is in the contra-variant position.

Similarly use annotations may be combined: $\tau.\{R_1\}.\{R_2\}$ is equivalent to $\tau.\{R_1 \cup R_2\}$.

Such equalities can be applied in any order such that: $((\tau.\{R_1\}\ \mathbf{at}\ r_1).\{R_2\})\ \mathbf{at}\ r_2$ can be rewritten as: $\tau\ \mathbf{at}\ r.\{R_1 \cup R_2\}$

6.4 Semantics

This section details the static semantics of region programs. These semantics follow the same course as the delayed typing static semantics. Each expression may be assigned a set of types.

The region semantics for the simple lambda-calculus based on those given in section 5.4 must consider three situations specifically.

- Expressions which create values stored in memory.
- Expressions which access memory directly.
- The region annotated expressions.

All other kinds of expressions may follow a region convention. Recall from the delayed typing static semantics that a delayed type represents a set of concrete types. The set of inference rules defining the static semantics given in this chapter also associates expressions with sets of types.

The main idea is that expressions may be typed with any region annotations and only at region annotated expressions does the type inference enforce the region rules. So for example a record creation expression may be associated with the set of all appropriate record types with any region annotations. This corresponds with storing the record in any region. If that record creation expression occurs within a **let region** expression then the typing of the **let region** expression will exclude from the set of inferable types the inappropriate region typings of the record creation expression. If the record creation expression occurs as the left hand side of an **at** expression, then the typing of the **at** expression will exclude all types which do not place the record in the given region. Similarly if the record creation expression is assigned to some identifier which is then used on the left hand side of an **at** expression then the inappropriate typings for the identifier are removed.

$\boxed{C \vdash expr \Rightarrow \overline{\tau}}$

Expressions which create data to be stored in a region in memory must be typed so as to indicate which region the value is stored in. The region the value is stored within must then be accessed when creation occurs and hence the type of the expression will contain be enclosed in a Uses region type with that region in the set of regions which are accessed. There are two expressions which create data, record creation expressions and function expressions which must create a function closure. As discussed above the rule for record creation allows the inference of all regions for the region in which the record is to be stored. The set in the following rule will therefore instantiate v to all regions. This set may later be pruned when an **at** expression is typed. The same is true for the function creation rule.

$$\frac{C \vdash \{fields\} \Rightarrow \overline{\tau}}{C \vdash \{fields\} \Rightarrow \{\tau \textbf{ at region}.(v).\{v\} \mid \tau \in \overline{\tau}\}} \tag{150}$$

$$\frac{\overline{\tau} = \{(\tau_1 \rightarrow \tau_2) \mid C_x[x \mapsto \{\tau_1\}] \vdash e \Rightarrow \overline{\tau}' \wedge \tau_2 \in \overline{\tau}'\}}{C \vdash \textbf{fun } x \rightarrow e \Rightarrow \{\tau \textbf{ at region}.(v).\{v\} \mid \tau \in \overline{\tau}\}} \tag{151}$$

The only expressions which directly access memory are record field access expressions and application expressions (which must access a function closure). Note that in this rule τ_1 may itself be an **at** type with region uses.

$$\frac{C \vdash e \Rightarrow \overline{\tau}}{C \vdash e.lab \Rightarrow \{\tau_1.\{s \cup v\} \mid \tau_1 = \mathcal{F}(\tau, lab) \wedge \tau \textbf{ at region}.(v).\{s\} \in \overline{\tau}\}} \tag{152}$$

The result type of an application must include the region in which the function closure is stored as being accessed. This rule will be updated to include the regions which may be accessed by the expressions e_1 and e_2 in section 6.4.2.

$$\frac{C \vdash e_1 \Rightarrow \overline{\tau}_1 \qquad C \vdash e_2 \Rightarrow \overline{\tau}_2}{C \vdash e_1\ e_2 \Rightarrow \{\tau.\{v\} \mid (\tau_2 \rightarrow \tau) \textbf{ at } v \in \overline{\tau}_1 \wedge \tau_2 \in \overline{\tau}_2\}} \tag{153}$$

The typing of the region annotated expressions is now detailed. The rules for the data creating expressions, rules 150 and 151, mean that any expression annotated by an **at** annotation may have in the set of types which may be inferred for it, all region annotations. Therefore the typing for the **at** expression needs only to remove from that set any which are incompatible with the given region. Whether or not this annotated expression actually accesses the given region is dependent on the kind of expression. It may for example be a variable which was a function argument. In this case the expression does not access the given region.

$$\frac{C \vdash e_1 \Rightarrow \overline{\tau}}{C \vdash e_1 \textbf{ at } v \Rightarrow \{\tau \textbf{ at region.}(v) \in \overline{\tau}\}} \tag{154}$$

The most complicated expression to type is the **let region** expression. This must invalidate the type of an expression which must not be accessed because the region is being removed from scope. It must remove from the set of typings all those types which are associated with a value stored in the region being removed from scope. Additionally all function types for which the body expression accesses the given region must also be invalidated. The function *noT* performs this operation, this is described in section 6.6.

$$\frac{C \vdash e_2 \Rightarrow \overline{\tau}}{C \vdash \textbf{let region } r \textbf{ in } e_1 \Rightarrow \{noT(r,\tau) \mid \tau \in \overline{\tau}\}} \tag{155}$$

6.4.1 Subsumption

The subsumption rules allow the inference of a less general type. That is a type which may be used in fewer situations. For regions there are subsumption rules required for the two extra region types; the **at** type and the Uses annotation type.

For the **at** type; where a type τ **at** r_1 may be inferred for a given expression then it is also safe to infer τ **at** r_2 if r_2 is a region which does not outlive r_1. This is because if the type system ensures that the value computed by the expression is not accessed after the region r_2 is deleted then it will also not be accessed after the region r_1 is deleted since r_1 is guaranteed to outlive r_2. This is expressed in rule 156. This rule uses the ordering over regions where $r_2 < r_1$ means that region r_1 outlives region r_2.

$$\frac{C \vdash e \Rightarrow \overline{\tau} \qquad \tau \ \mathbf{at}\ r_1 \in \overline{\tau} \qquad r_2 < r_1}{C \vdash e \Rightarrow \{\tau \ \mathbf{at}\ r_2\} \cup \overline{\tau}} \tag{156}$$

For the uses annotation if the type system can infer the type $\tau.\{R_1\}$ for a given expression, then it is safe to allow the type $\tau.\{R_2\}$ to be inferred for the same expression provided that $R_1 \subset R_2$. Since if the type system guarantees that the expression will not be evaluated outside the scope of the regions in R_2 then it will also not be evaluated outside the scope of the regions in R_1. This is expressed in rule 157.

$$\frac{C \vdash e \Rightarrow \overline{\tau} \qquad \tau.\{R_1\} \in \overline{\tau} \qquad R_2 \supset R_1}{C \vdash e \Rightarrow \{\tau.\{R_2\}\} \cup \overline{\tau}} \tag{157}$$

The **no** type may be given to any expression which is otherwise typable. This allows us to give the **no** type to expressions whose region is then removed from scope. For example the expression **let region** r **in** e **at** r may be given the **no** type since the inner expression e may first be typed as the set $\overline{\tau}$. Using rule 154 this is then given the set $\overline{\tau}_1$ such that all types have the appropriate **at** annotation. Using this new subsumption rule it is now given the set $\{\mathbf{no}\} \cup \overline{\tau}_1$. Using the rule for **let region** expressions 155 this is reduced to the singleton set containing only the **no** type.

$$\frac{C \vdash e \Rightarrow \overline{\tau}}{C \vdash e \Rightarrow \{\mathbf{no}\} \cup \overline{\tau}} \tag{158}$$

6.4.2 Region Convention

The remaining expressions are typed in the same manner as the original delayed typing scheme given in section 5.4. However the typing rules must allow for the use of regions within sub-expressions. Recall the rule for function application Rule 153, this must now be updated to the form:

$$\frac{C \vdash e_1 \Rightarrow \overline{\tau}_1 \qquad C \vdash e_2 \Rightarrow \overline{\tau}_2}{C \vdash e_1\ e_2 \Rightarrow \{\tau.\{v \cup R\} \mid ((\tau_2 \rightarrow \tau)\ \mathbf{at}\ v).\{R\} \in \overline{\tau}_1 \wedge \tau_2.\{R\} \in \overline{\tau}_2\}} \tag{159}$$

This rule means: a type may be in the set of types inferred for this application expression if the appropriate arrow type is in the set inferred for the function expression and the appropriate argument type is in the set of types inferred for the argument expression. Additionally the uses annotation on the inferred type must be the union of the set of regions which may be accessed when computing the function expression and the argument expression. This may be enforced by requiring that both uses annotations are the same. The subsumption rule 157, can be used to inflate either of the uses annotation to the union of the two.

Additionally note that the return type for the function τ may itself be of the form $\tau.\{R_1\}$. Because we ensure that $\tau.\{R_1\}.\{R\}$ is equivalent to $\tau.\{R \cup R_1\}$, it must be that any type in the set inferred for the application expression contains all those regions which may be accessed by the evaluation of the function body.

All of the remaining expression rules must be updated in this fashion. A region convention is used for this purpose. The region convention states that where a rule is presented as:

$$\frac{C \vdash \mathit{phrase}_1 \Rightarrow \overline{\tau}_1 \qquad \ldots \qquad C \vdash \mathit{phrase}_n \Rightarrow \overline{\tau}_n}{C \vdash \mathit{phrase} \Rightarrow \{\tau \mid \tau_1 \in \overline{\tau}_1 \ldots \tau_n \in \overline{\tau}_n\}}$$

This is instead required to mean:

$$\frac{C \vdash \mathit{phrase}_1 \Rightarrow \overline{\tau}_1 \qquad \ldots \qquad C \vdash \mathit{phrase}_n \Rightarrow \overline{\tau}_n}{C \vdash \mathit{phrase} \Rightarrow \{\tau.\{R\} \mid \tau_1.\{R\} \in \overline{\tau}_1 \ldots \tau_n.\{R\} \in \overline{\tau}_n\}}$$

6.5 Notes

The record creation typing rule (Rule 150) already accounts for the case that the field initialising expressions access some regions. In this case the type of the record will have type $\tau.\{R\}$. So the types in the set inferred for the record creation will all have the form $\tau.\{R\}$ **at** $v.\{v\}$, which by the rule of combining region annotation types is equivalent to: τ **at** $v.\{v \cup R\}$.

6.6 The region scoping function

The region scoping function *noT* must invalidate the type of values which are stored in regions which are leaving scope. Additionally functions which may access such regions must also be invalidated as it would be unsafe for them to be run.

Notice the final line in this function. It states that if the type of a value returned by the **let region** expression contains the region which is leaving scope in its uses annotation then we can remove it from there. The reasoning is that if the computation of that value will occur whilst that region is in scope then any future checks need not re-check this property. This is not necessary since any future checks will not concern the currently leaving scope region however it does allow the type system to clean up the inferred types.

$$\begin{aligned}
noT(v, \mathbf{int}) &= \mathbf{int}\\
noT(v, \mathbf{bool}) &= \mathbf{bool}\\
noT(v, \tau\ \mathbf{at\ region}.(v)) &= \mathbf{no}\\
noT(v, (\tau_1 \rightarrow \tau_2.\{v\})) &= \mathbf{no}\\
noT(v, \tau.\{R\}) &= \tau.\{R/v\}
\end{aligned}$$

All other types are operated on recursively.

6.7 Conclusions

The region system for memory management provided a convenient solution to the problem of memory management for Nitro. Firstly Nitro did not wish to require a common internal representation, since this would conflict with the major goal of the foreign data interface which was to interface directly with foreign data. By using a conservative garbage collector which operates on ambiguous roots this could be avoided without the need to resort to anything other than automatic runtime garbage collection. However if this is used then the programmer is not able to employ separate memory management schemes for the programs which Nitro is used to interface with and in particular provide an abstraction for. As discussed in the background section of this chapter important properties such as predictability and control are, at least to some extent, lost.

With the region memory management scheme a Nitro program is able to allocate foreign values and either collect them automatically with the use of the region scheme or allow them to be collected automatically in the special heap region. Furthermore Nitro can be used to implement a different form of garbage collection for the foreign values with which it is interfacing. This is done by abstracting the type of values such that the arrays holding them can be updated with new values. In this way memory is re-used by the calling program with the aid of the Nitro implemented abstraction.

The delayed typing scheme was used to allow accurate typing of Nitro's region constructs. Code that made no use of regions does not necessarily need to be polluted with region variables and annotations on the types of such values. This is one area in which further work could achieve even better results by using default annotations, such as a higher order function having the same region effects as the sum of all of its functional arguments. A web based demonstration of the delayed typing of region constructs can be found at `http://homepages.inf.ed.ac.uk/s9810217/region_demo.html`.

This chapter has brought together the foreign data interface and the delayed typing scheme to fill in the missing component of the abstraction-level programming language Nitro. The Nitro programmer is now able to:

- Directly manipulate foreign values.
- Embed foreign type information into Nitro types.
- Control the lifetimes of those foreign values.
- Relinquish control of the lifetimes of those foreign values.

This last is important, for some abstraction-level tasks, in particular when augmenting an existing abstraction, the programmer does not want to take over the control of the allocation details of the foreign values. The OCaml equality operator example of section 4.6.1.6, should not manage the returned boolean value itself but instead register it with the existing OCaml runtime garbage collector.

The advantages of using Nitro to perform these tasks are:

- The type system can be used to make guarantees about the behaviour of the abstraction-level code.
- The programmer has the use of high-level productivity increasing language constructs.
- The programmer may write their abstraction code in a language much closer to the language for which the abstraction is provided, which is often familiar to the abstraction-level programmer.
- The programmer can manipulate the strengths and subjects of the guarantees provided by the Nitro type system.

This last advantage again requires further explanation. It was shown in the OCaml equality operator that the type system could ensure that a valid OCaml value was returned. With the use of the delayed typing system and the subtyping relations it provides, it is also possible to ensure that not only a valid OCaml value, but a valid OCaml boolean is returned.

Finally the drawbacks of using Nitro with a region based memory management scheme are:

- Occasionally the type system can mean that a task solvable in C in an obvious way is made more difficult by the need to satisfy the Nitro type system although often the gain in maintainability of the resulting code can be worth the extra effort.

Chapter 7

Conclusions

This chapter summarises the conclusions made within the current thesis. It begins with a discussion of the Cyclone language with respect to the Nitro language. Following this the main contributions made within this thesis and through the development of the Nitro language are discussed and finally some future work is proposed.

7.1 Cyclone discussion

This thesis has talked about providing the abstraction-level programmer with features predominately found in higher-level languages. Cyclone[18], a safe variant of the C programming language, is a related language. Much research and effort has gone into the design of the Cyclone language and the development of the associated programming tools.

A couple of interesting questions arise; why not create a functional version of Cyclone? and why not compile Nitro to Cyclone?

7.1.1 A functional Cyclone

Cyclone has already been designed to create a safe low/abstraction-level programming language by starting with a low-level language C, and adding high-level features and restricting unsafe low-level features. Nitro attempts to come from the other direction by starting with a high-level type-safe functional language and adding in support for abstraction-level tasks. In doing so one feature of Nitro is that application programming can be undertaken in Nitro without any use of the abstraction-level facilities. This presents the application developer with a good choice, if they foresee that their application may require abstraction-level components, for example interfacing with a legacy library. Such foresight has in the past led to developers using an unsuitably low-level language to implement an application knowing that using a high-level language would require the writing of marshalling routines. Writing the application in Nitro means that no marshalling routines are required and therefore performance worries need not induce the selection of an inappropriate low-level language.

7.1.2 Compiling Nitro to Cyclone

The decision to design a new low-level language as opposed to providing a variant of Cyclone does not automatically prohibit the possibility of compiling the new language Nitro to the imperative language Cyclone. This is in contrast to the approach of compiling Nitro directly to assembly language or compiling into C code thus using the C compiler as a high-level assembler. There are a few good reasons for considering this. Firstly the type checking performed by the Cyclone compiler is stricter than that done by the C compiler. This means that there can be greater confidence in the correctness of the Nitro compiler since its output Cyclone program is more thoroughly checked by the Cyclone compiler than the equivalent C output program would be checked by the C compiler.

Secondly, Cyclone already has a region memory management implementation. In particular there is a region inference implementation which attempts to compute the best regions into which to place objects. Currently Nitro has no such system, a Nitro programmer must therefore either rely on traditional runtime garbage collection or manually choose the regions in which to place their objects. A region inference system, such as described in [95, 98], for Nitro would require a significant investment of time, gaining one for free via compiling to Cyclone would certainly be desirable.

There are other benefits to compiling Nitro to Cyclone; Cyclone already implements some other features that must be written in the Nitro compiler such as pattern matching, exceptions and tagged union types.

There are two major reasons for not compiling to Cyclone. The first is the engineering effort involved and the second is the restrictions on the facilities that Nitro could provide because each such facility must be converted into equivalent Cyclone code. Most of the points that follow here fall into one or both of these major reasons.

The delayed typing scheme discussed in chapter 5 was in part developed to allow the type checking of code that was previously not well typed. This meant that a larger number of safe programs could be allowed to pass type checking and be compiled. It is not clear that such programs could be translated into Cyclone code which would fit into the Cyclone type system and if so whether such a translation would require a lot of implementation effort or result in an inefficient compilation scheme.

Cyclone cannot access all of C's data structures and one must occasionally write a wrapper to convert between the C representation and the Cyclone one and back again. It was a major aim of the Nitro foreign data interface to avoid such marshalling routines.

Compiling Nitro to C involves the use of C as an extended assembler. This means back-end issues such as register allocation and low-level optimisations which are not relevant to the Nitro language can be avoided. The Nitro compiler does not translate Nitro types to C types, all Nitro values are declared with type `nitro_value_t`. To create tuple or record types then, we cast the tuple value to a `nitro_value_t` pointer and assign the fields using array subscripting. The C compiler can be used in this way, as a higher-level assembler, because the type system it provides is a weak type system.

To compile to Cyclone the type system cannot be abused in such a manner, and the Nitro compiler would be required to translate all Nitro types into equivalent Cyclone types. Therefore the compilation would be an intelligent translation rather than a compilation. Such a translation would require a large investment of time.

The region inference of the Cyclone programming language would not necessarily fit the expectations of a region inference scheme for Nitro. That is, the region annotations that could be inferred of a Nitro program will not necessarily correspond to those regions that will be inferred for the same objects in the resulting Cyclone program. In fact there are occasions when the Cyclone region inference algorithm fails and the user must supply a region annotation. Whether such cases could be avoided or whether the Nitro compiler could automatically provide such annotations would depend upon the translation from Nitro to Cyclone. In any case it is debatable whether or not compiling Nitro to Cyclone does indeed give us a region inference system 'for free'.

Finally, while this does not affect either the facilities Nitro can provide, or the engineering effort involved in the compiler, the Cyclone runtime system - in particular that which maintains the regions used by the program - is written in C. A goal of the Nitro language was that the runtime system be itself written entirely in Nitro and this was achieved in the implementation provided. Compiling to Cyclone would leave three choices:

1. The Cyclone runtime is not used thereby losing automatic region inference and a major advantage of compiling to Cyclone.

2. Alternatively the hope that a Nitro region runtime would be used could be sacrificed.

3. Finally the Cyclone region runtime could be re-implemented in Nitro. This final option would again require significant implementation effort.

7.2 Abstraction-Level Programming

In this thesis the term *abstraction-level* programming was used to further distinguish kinds of programming tasks. Initially two kinds were identified as high-level and low-level. The high-level tasks were defined to be the implementation of algorithms in a hopefully machine-independent manner, using the abstractions to the particular machine provided. Those abstractions were provided by the low-level programmer. The kinds of tasks done in low-level programming were then further categorised into low-level and abstraction-level programming tasks. The abstraction-level programming tasks deal purely with the representation and management of data, while the low-level tasks are those in which direct access to the internals of the machine are required. It was argued that there was a gap in programming language design where most languages were designed as either high-level; offering full abstraction from the machine, and low-level; offering full access to the machine. In between these two extremes an abstraction-level programming language would be one that offered access to and control of the data representation but retained some high-level language features such as type-safety.

7.3 Evaluation

In this section Nitro is evaluated as a programming language designed to facilitate abstraction-level programming.

7.3.1 Accessing Foreign Data

Chapter 4 dealt mostly with the accessing of foreign data from within Nitro. Type definitions were used to convey to the compiler how data values are represented in memory. It was shown that many important data types could be represented in this way. The compiler's type system was modified to support the ability for the programmer to dictate within a type definition constraints on the use of a value. This was most obvious when using **immediate** data types in which the arguments of overlapping constructors were incompatible. The quintessential example is that of a C data structure in which the NULL pointer value represents some choice. As in:

```
type immediate c_string =
    Null_string { 0 }
  | Cstring { _ } of c_char_array
```

As discussed in section 4.2.5.2, because the tag values overlap, and the arguments are not compatible, whenever a value of type `c_string` is matched against, the constructor `Null_string` must be matched against before the constructor `Cstring`. The equivalent definition in C relies upon the programmer always checking for the null string where appropriate. Note as well that whenever the value is guaranteed not to be the null string, for example after this match has been performed, then the value has type `c_char_array` and hence the programmer does not needlessly check for the null string. Also, although this layout cannot be defined in a high-level language, it can be emulated with a union type. The equivalent definition using a union type would be forced to use a block in memory, with the first value used to determine the presence of a following C string pointer. So the Nitro definition combines the best of both worlds, the programmer retains the efficiency in both space and time of the C representation and the guarantee that one cannot misuse the pointer of the high-level union type definition.

7.3.1.1 Limitations

There are certain kinds of data that Nitro is not yet capable of representing. A null terminated array for example is awkward to represent. A possible definition is:

```
type null_terminated
    Terminate { 0 }
  | Non_term { _ } precedes null_terminated
```

This allows us to examine a null-terminated array such as those produced by C language libraries. The size of a value can be adjusted to allow access to a null-terminated C style string as well. A **`precedes`** argument is handled specially by the compiler such that it can create a memory block large enough to occupy both the argument and the tag. This works if the argument is created at the time of the constructor but with a recursive **`precedes`** argument, the argument will, at least some of the time, be a reference to an existing value. This whole value then must be copied across by the code produced by the Nitro compiler. Hence building up a large `null_terminated` value may be quite inefficient.

There are three solutions to this problem, either accept that these kinds of values are awkward to represent and rely on external procedures to create values of this kind or represent a value as an ordinary Nitro bare array and rely on the programmer to ensure that the final value is the NULL value. Finally add in language support for null terminated arrays as is done in the Cyclone language.

The first and second options here are almost the same, because in the external procedure again the programmer is relied upon to ensure that the array is indeed terminated with a NULL character. It could be argued that external libraries are more thoroughly tested and hence this solution is more desirable, however it could also be argued that within Nitro the array cannot be illegally accessed outside of its bounds whether or not the NULL character is in place. Note that in either case the Nitro code cannot illegally access the null-terminated array, unless the first option is used and the external library is faulty. The final solution suffers none of the previous complications at the cost of complicating both the language and the compiler.

7.3.1.2 Overall

In general the foreign data interface of Nitro allows the functional programmer direct access to foreign data in a familiar environment. This ability to control the representation of data can also be used to optimise the private data structures in a Nitro program. The most important uses however are facilitating the writing of language routines which cannot be done in the host language because it prevents access to the representation of values, and the marshaling of data from foreign or legacy libraries into a format manageable by the programmer's preferred high-level language. Chapter 4 has shown that using Nitro to provide such routines can result in more readable, maintainable and trustworthy code.

7.3.2 Delayed Typing

This section evaluates the effect of modifying the type system of Nitro to include a delayed type scheme. Although in chapter 5 many advantages were discussed such as the ability to infer exception annotations and the ability to remove some of the slack of the type system, the main advantage was the increased accuracy of the type system and the provision for sub-typing.

Allowing sub-typing was shown to allow the programmer to encode foreign type information within the Nitro type. In particular a value could be seen at a type which indicated it was some value produced by a particular programming language such as OCaml. At the same time, the same value could be seen at a Nitro type which indicated that it was of a particular (OCaml) type of OCaml value. This meant that it was possible to ensure that the value returned from a routine to an OCaml program was of, say, the OCaml `boolean` type but the same value was also of the more general OCaml value type and hence could, for example, be registered with the OCaml garbage collector.

Delayed typing has been mostly a success in the goals that were identified. The cost is in the lack of conciseness within the inferred types. Type constraints can and arguably should always be given, when used with regular use of type constraints the delayed typing system allows the user further expressive power to indicate via the type, information concerning the values in the program. Because Nitro is used as an abstraction level programming language, such information within the type is highly desirable because an abstraction can be seen as the implementation of an interface and hence the more that interface can be described mechanically in the form of a signature the better.

The extra accuracy within the type is also required to allow more programs to be type checked. This is particularly important for an abstraction-level programming language because the code is generally performing some task that cannot be type checked within the high-level language utilising the abstraction provided. Finally the extra accuracy of the type can also be used by both the programmer and the compiler to optimise code. Checks can be omitted by the compiler using the extra type information to ensure that the check can only ever evaluate in one direction. The programmer can use extra type accuracy to optimise data structures notably when used in conjunction with the foreign data interface.

7.3.3 Regions

A region memory management scheme was given to Nitro to allow abstraction-level programs which cannot use a runtime garbage collector to be written. Region memory management is no silver bullet and using such a scheme has shown how hard a problem the correct deployment of memory management is. There are many different schemes to choose from, some which require language support and others which do not. Selecting a memory management scheme which does require language support involves a small amount of risk since it may be that a better solution will be discovered in the future, thus code which uses the language support must be modified. However the region scheme works well in tandem with a conservative garbage collector. The programmer can control only those data value lives which need to be controlled by the program or that the programmer feels can benefit from such explicit control.

7.4 Trusted applications

In the introduction several places of trust were identified. These places of trust are distinguished from each other in two respects, firstly in how and what they are trusted and/or checked for and secondly in their relationships towards each other.

The identified places of trust are:

- The application code

- The Compiler
- The Abstraction Code
- The Compiler-Abstraction Code Link

There were two separate properties of code that must be trusted, the safety of the code and the correctness. The safety of code referred to the possibility that it may attempt to make an illegal access into memory, in broad terms this meant that if a given memory location contains a value of a given type, then that memory location must not be accessed as though it stored a value of an incompatible type. Correctness is much harder to define, however a very generic definition would say that code which evaluates all inputs to expected results is code which is correct.

It was noted that the safety and correctness of the entire application were linked, for the safety of one part of the trusted application depends upon the correctness of another part. For example the safety of the application code itself depends upon the correctness of the compiler in translating safe and correct code to equivalent safe and correct machine code.

The main contribution of this thesis has been to describe the development of a language; Nitro, designed to increase the confidence in the abstraction code. This is done by providing the programmer with features found in high-level languages used to code application code, and supplementing this with low-level features required for writing such abstraction code. Once the application and abstraction code are combined into a single machine program the user can be more confident in the safety and correctness of the whole program because Nitro provides greater guarantees than a low-level programming language would.

7.5 Contributions and Further Work

The main contributions of this thesis have been:

- A formal definition of the core of the Nitro functional programming language.
- An extension to the formal definition to incorporate direct manipulation of foreign langauge objects.
- The development of a the delayed-typing scheme.
- An extension to the formal defintion to incorporate the delayed-typing scheme.
- Demonstration of some properties of the delayed-typing scheme.
- An extension of the formal definition to include facilities for managing the lifetimes of objects through use of a region memory management scheme.
- The full implementation of the Nitro programming language including the extensions noted.

The remainder of this section highlights some of the future work that could be undertaken to improve the results contained within this thesis.

7.5.1 Foreign Data Interface

The foreign data interface allows a variety of control over the representation of data whilst still allowing the compiler to restrict the programmer to the use of data types for which consistent access can be guaranteed. This means that the programmer can use as much space as is actually required to distinguish the distinct values within their types. What the programmer cannot do is use external information which is not available within the data itself.

This inability is highlighted when the data is somehow previously analysed. A good example is the implementation of a virtual machine which interprets a stream of byte-code instructions. One such instruction may have the meaning; take the value on top of the stack and dereference it as a pair.

Another instruction might have the meaning, take the top two values of the stack and add them together as integers. If the stack is represented as a list of arbitrary values a C implementation can cast the top of the stack according to what the next instruction expects it to be. This is of course an unsafe thing to do. However if the instruction stream is first run through a verifier, and that verifier is trusted to be correct, then all of the casts may be trusted to be safe. Such a simple encoding in Nitro would not be possible. Future work into this area could yield exciting results not just for abstraction-level programming, but for low-level typing in general.

7.5.2 Delayed Types

There are many other applications of the delayed typing scheme. One that I would particularly like to add would be the application to uniqueness typing. In uniqueness typing a name is typed as being either the sole reference to that value or one of potentially many. When this is done destructive update may be an allowable and useful addition to the language. The Clean language is a lazy functional language which allows side effects using uniqueness typing, "worlds" which are updated must be updated through a unique reference to that world.

Other possible uses for delayed typing include:

- Type classes, due to the ability to allow sub-typing constraints.
- Many more annotations, in this thesis two such annotations were region and exception effects. Other annotations could include destructive update, uniqueness, use of foreign and possibly unsafe code, author signature and concurrency constraints.
- Other analyses such as space and time resource bounds, since such analyses can utilise the arguments to a function retaining the delayed type of a function could be useful.

- The use of the extra information contained within a delayed type to give a more helpful error message.

Future work in applying a delayed typing scheme to these typing areas is an interesting possibility. Finally for delayed typing a possible complaint is that the more verbose types which are inferred are less readable than those inferred by a non-delayed type inference system. It was noted that once annotations of just one kind are added the delayed type system can often result in a more readable type. Future work in how best to report types which contain annotations and results of analyses such as those listed above could enhance this situation.

7.5.3 Regions

Once a language has introduced explicit region constructs as a method of allowing the programmer safe control over aspects of memory management the possibilities for future invention become large. Many distinct kinds of regions exist and many have been developed for use in the Cyclone language. A good overview can be found in [97, 81]. In particular most general memory management schemes can be translated into a type of region, thus the programmer has many separate memory management schemes available to them within the same program. Different values can be managed differently depending upon how the programmer decides that the lifetime of the value will behave during execution. Region types already considered include, but are not limited to, the following:

- Dynamically collected heap region
- A stack region, held on the machine's stack and automatically deleted at the termination of the function in which it was created.
- A unique region in which values stored have only one reference to them.
- A reference counted region.

Future work done here would benefit other systems which utilise a region memory management scheme including Cyclone. The future work in relation to Nitro would therefore include:

- Use substantial testing to establish the efficiency and scalability of the current Nitro scheme including the current runtime, itself written in Nitro.
- Investigate the translation of the current region inference techniques as used for SML [99] and Cyclone.
- Extend the current Nitro scheme to include those kinds of regions already successfully integrated into the Cyclone language.

7.5.4 Modules

In this thesis Nitro has been described without a module system. The interesting part in adding a module system to Nitro is in how it would cooperate with the delayed typing system. Delayed typing is arguably best used to infer a very generic but accurate type which the programmer may restrict in a very specific way to the exact type required. Where module signatures are used the signature can describe a very precise interface which it is then possible to implement in many ways. This is because using sub-typing constraints and type annotations such as region or exception annotations, the signature given can describe the behaviour desired with a high degree of precision. However because of the very general types inferred the signature can be matched with a wide range of implementations.

Bibliography

[1] David Evans and David Larochelle. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Softw.*, 19(1):42–51, 2002.

[2] S. Johnson. Lint, a C program checker. *Unix Programmer's Manual*, 2, 1978.

[3] Ken Ashcraft and Dawson Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 143, Washington, DC, USA, 2002. IEEE Computer Society.

[4] J. J. Hallett and Assaf J. Kfoury. Programming Examples Needing Polymorphic Recursion. *Electr. Notes Theor. Comput. Sci.*, 136:57–102, 2005.

[5] Robert Cartwright and Mike Fagan. Soft typing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292, New York, NY, USA, 1991. ACM Press.

[6] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[7] American National Standards Institute. *ANSI/ISO/IEC 9899-1999: Programming Languages — C*. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1999.

[8] Stefan Monnier and Zhong Shao. Typed Regions. Technical Report YALEU/DCS/TR-1242, Dept. of Computer Science, Yale University, New Haven, CT, October 2002.

[9] Daniel C. Wang and Andrew W. Appel. Type-preserving garbage collectors. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 166–178, New York, NY, USA, 2001. ACM Press.

[10] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, 1997.

[11] X. Leroy, J. Vouillon, D. Doligez, et al. `http://caml.inria.fr`, 1996-2007. The Objective Caml system. Software and documentation available on the Web.

[12] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. A principled approach to operating system construction in Haskell. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 116–128, New York, NY, USA, 2005. ACM Press.

[13] Richard B. Kieburtz. P-logic: property verification for Haskell programs. ftp://ftp.se.ogi.edu/pub/pacsoft/papers/Plogic.pdf, 2002.

[14] Edoardo Biagioni, Robert Harper, and Peter Lee. A Network Protocol Stack in Standard ML. Submitted for publication to *Higher-Order and Symbolic Computation*.

[15] Herb Derby. The performance of FoxNet 2.0. Technical Report CMU-CS-99-137, School of Computer Science, Carnegie Mellon University, June 1999.

[16] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley Longman, Reading, MA, third edition, 1997.

[17] ECMA. *ECMA-334: C# Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, second edition, December 2002.

[18] T. Jim, G.Morrisett, D.Grossman, M.Hicks, J.Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. In *USENIX Annual Technical Conference*, Monterey, CA, USA, 2002.

[19] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 128–139, New York, NY, USA, 2002. ACM Press.

[20] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in the real world. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 232–244, New York, NY, USA, 2003. ACM.

[21] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.

[22] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. *SIGPLAN Not.*, 29(6):290–301, 1994.

[23] Jeffrey S. Foster. *Type Qualifiers: Lightweight Specifications to Improve Software Quality*. PhD thesis, University of California, Berkeley, December 2002.

[24] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting Format-String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, pages 201–220, August 2001.

[25] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2002. ACM Press.

[26] Satish Chandra and Thomas Reps. Physical type checking for C. *SIGSOFT Softw. Eng. Notes*, 24(5):66–75, 1999.

[27] Geoffrey Smith and Dennis Volpano. A sound polymorphic type system for a dialect of C. *Science of Computer Programming*, 32(1–3):49–72, 1998.

[28] Matthias Blume. No-Longer-Foreign: Teaching an ML compiler to speak C "natively". *Electr. Notes Theor. Comput. Sci.*, 59(1), 2001.

[29] Michael Furr and Jeffrey S. Foster. Checking type safety of foreign function calls. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 62–72, New York, NY, USA, 2005. ACM Press.

[30] Thorsten Brunklaus and Leif Kornstaedt. A Virtual Machine for Multi-Language Execution. Technical report, Saarland University Computer Science Programming Systems, November 2002.

[31] Leif Kornstaedt. Alice in the land of Oz – an Interoperability-based Implementation of a Functional Language on Top of a Relational Language. In *Proceedings of the First Workshop on Multi-language Infrastructure and Interoperability (BABEL'01), Electronic Notes in Computer Science*, volume 59, Firenze, Italy, September 2001. Elsevier Science Publishers.

[32] ECMA. *ECMA-335: Common Language Infrastructure (CLI)*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, third edition, June 2005.

[33] D. Syme and J. Margetson. The F# website. See `http://research.microsoft.com/fsharp/`, 2006.

[34] Nick Benton, Andrew Kennedy, and Claudio V. Russo. Adventures in interoperability: the SML.NET experience. In *PPDP '04: Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 215–226, New York, NY, USA, 2004. ACM Press.

[35] Nick Benton, Andrew Kennedy, and George Russell. Compiling standard ML to Java bytecodes. In *ICFP '98: Proceedings of the third ACM SIG-*

PLAN international conference on Functional programming, pages 129–140, New York, NY, USA, 1998. ACM Press.

[36] Nick Benton and Andrew Kennedy. Interlanguage working without tears: blending SML with Java. In *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 126–137, New York, NY, USA, 1999. ACM Press.

[37] Kathleen Fisher and John Reppy. The design of a class mechanism for Moby. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 37–49, New York, NY, USA, 1999. ACM Press.

[38] Kathleen Fisher, Riccardo Pucella, and John H. Reppy. A Framework for Interoperability. *CoRR*, cs.PL/0405084, 2004.

[39] My Hoang and John C. Mitchell. Lower bounds on type inference with subtypes. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 176–185, New York, NY, USA, 1995. ACM Press.

[40] Patrick Lincoln and John C. Mitchell. Algorithmic aspects of type inference with subtypes. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 293–304, New York, NY, USA, 1992. ACM Press.

[41] Harry G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 382–401, New York, NY, USA, 1990. ACM Press.

[42] D. Rémy. Type checking records and variants in a natural extension of ML. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 77–88, New York, NY, USA, 1989. ACM Press.

[43] M. Wand. Type inference for record concatenation and multiple inheritance. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 92–97, Piscataway, NJ, USA, 1989. IEEE Press.

[44] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *J. Funct. Program.*, 17(1):1–82, 2007.

[45] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18(9):807–820, 1988.

[46] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In Yves Bekkers and Jacques Cohen, editors, *IWMM*, volume 637 of *Lecture Notes in Computer Science*, pages 1–42. Springer, 1992.

[47] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *IWMM '95: Proceedings of the International Workshop on Memory Management*, pages 1–116, London, UK, 1995. Springer-Verlag.

[48] Xavier Leroy, November 2006. personal communication via e-mail.

[49] R. Milner, November 2006. personal communication via e-mail.

[50] Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 26–37, New York, NY, USA, 2001. ACM Press.

[51] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML: Revised 1997.* The MIT Press, 1997.

[52] Andreas Rossberg. Defects in the Revised Definition of Standard ML. Technical report, Saarland University, Saarbrücken, Germany, October 2001. Updated 2004/06/22, 2005/01/13, 2005/01/26, 2006/07/18., 2007/01/22.

[53] T Nordin and Simon Peyton Jones. Green Card: a foreign-language interface for Haskell. In *Proceedings of the Haskell Workshop 1997*, June 1997.

[54] Thien-Thi Nguyen, Loic Dachary, Oleg Tolmatcev, et al. `http://www.swig.org`, 2000-2007. The Swig interface compiler, software and documentation available on the web.

[55] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0 (Fourth Edition), September 2006. `http://www.w3.org/TR/xml/`.

[56] *System Application Program Interface (API) [C Language].* Information technology—Portable Operating System Interface (POSIX). IEEE Computer Society, 345 E. 47th St, New York, NY 10017, USA, 1990.

[57] Jacques Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering, Sassaguri, Japan.* Available from `http://www.math.nagoya-u.ac.jp/~garrigue/papers/fose2000.html`, 2000.

[58] Konstantin Läfer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Trans. Program. Lang. Syst.*, 16(5):1411–1430, 1994.

[59] Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 26–37, New York, NY, USA, 2001. ACM.

[60] TE Dickey, 1993-2007. Ncurses text user interface library available at http://www.gnu.org/software/ncurses/ncurses.html and also http://dickey.his.com/ncurses/ncurses.html.

[61] Michael I. Schwartzbach. Polymorphic Type Inference. Technical Report LS-95-3, brics, June 1995. viii+24 pp.

[62] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM Press, 1982.

[63] Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1):38–94, 1994.

[64] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[65] Brus T, Eekelen van MCJD, Plasmeijer MJ, and Barendregt HP. Clean - A Language for Functional Graph Rewriting. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture*, pages 364–384, Portland, Oregon, USA, 1987. Springer Verlag.

[66] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM.

[67] Per Martin-Löf. *Intuitionistic Type Theory*, volume 1 of *Studies in Proof Theory: Lecture Notes*. Bibliopolis, Napoli, 1984.

[68] Howgwei Xi and Frank Pfenning. Dependent Types in Practical Programming. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 214–227, New York, NY, 1999.

[69] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential types. In *POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 37–51, New York, NY, USA, 1985. ACM.

[70] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theor. Pract. Object Syst.*, 5(1):35–55, 1999.

[71] Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, February 1991.

[72] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, 1991.

[73] Wei-Ngan Chin and Siau-Cheng Khoo. Calculating sized types. In *PEPM '00: Proceedings of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 62–72, New York, NY, USA, 1999. ACM.

[74] P.M. Achten, J.H.G. Groningen van, and M.J. Plasmeijer. High-level specification of I/O in functional languages. In *Proceedings of Glasgow workshop on Functional programming*, pages 1–17, Ayr, Scotland, 1992. Springer-Verlag.

[75] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, January 1989.

[76] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type Classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, March 1996.

[77] John Peterson and Mark P. Jones. Implementing Type Classes. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 227–236, 1993.

[78] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *IWMM '95: Proceedings of the International Workshop on Memory Management*, pages 1–116, London, UK, 1995. Springer-Verlag.

[79] Andrew W. Appel. Garbage Collection can be Faster than Stack Allocation. *Information Processing Letters*, 25(4):275–279, 1987.

[80] Benjamin Zorn. The measured cost of conservative garbage collection. *Softw. Pract. Exper.*, 23(7):733–756, 1993.

[81] Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Experience with safe manual memory-management in cyclone. In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, pages 73–84, New York, NY, USA, 2004. ACM Press.

[82] Henry G. Baker. Lively Linear Lisp — 'Look Ma, No Garbage!'. *ACM SIGPLAN Notices*, 27(9):89–98, 1992.

[83] Oukseh Lee, Hongseok Yang, and Kwangkeun Yi. Static insertion of safe and effective memory reuse commands into ML-like programs. *Sci. Comput. Program.*, 58(1-2):141–178, 2005.

[84] Jeffrey M. Barth. Shifting garbage collection overhead to compile time. *Commun. ACM*, 20(7):513–518, 1977.

[85] Nancy Mazur, Peter Ross, Gerda Janssens, and Murice Bruynooghe. Practical Aspects for a Working Compile Time Garbage Collection System for Mercury. In *Proceedings of the 17th International Conference on Logic Programming*, pages 105–119, London, UK, 2001. Springer-erlag.

[86] F. Henderson, T. Conway, Z. Somogyi, and D. Jeffery. The Mercury Language Reference Manual, 1996. available from: http://www.cs.mu.oz.au/mercury.

[87] Martin Hofmann. A Type System for Bounded Space and Functional In-Place Update. *Nord. J. Comput.*, 7(4):258–289, 2000.

[88] David Aspinall and Martin Hofmann. Another Type System for In-Place Update. In Daniel Le Métayer, editor, *ESOP*, volume 2305 of *Lecture Notes in Computer Science*, pages 36–52. Springer, 2002.

[89] Donald Sannella, Martin Hofmann, David Aspinall, Stephen Gilmore, Ian Stark, Lennart Beringer, Hans-Wolfgang Loidl, Kenneth MacKenzie, Al-

berto Momigliano, and Olha Shkaravska. *Mobile resource guarantees*, volume 6, pages 211–226. Intellect, 2007.

[90] Fergus Henderson. Accurate garbage collection in an uncooperative environment. In *ISMM '02: Proceedings of the 3rd international symposium on Memory management*, pages 150–156, New York, NY, USA, 2002. ACM Press.

[91] Douglas T. Ross. The AED free storage package. *Commun. ACM*, 10(8):481–492, 1967.

[92] David Gay and Alexander Aiken. Memory Management with Explicit Regions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 313–323, 1998.

[93] David Gay and Alexander Aiken. Language Support for Regions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 70–80, 2001.

[94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 188–201, New York, NY, USA, 1994. ACM Press.

[95] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Trans. Program. Lang. Syst.*, 20(4):724–767, 1998.

[96] Mads Tofte, Lars Birkedal, Martin Elsman, Tommy Højfeld Olesen Niels Hallenberg, and Peter Sestoft. *Programming with Regions in the ML Kit (for Version 4)*. IT University of Copenhagen. April 2002, April 2002.

[97] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 282–293, New York, NY, USA, 2002. ACM Press.

[98] Jean-Pierre Talpin. A simplified account of region inference. Technical Report 4104, INRIA, Jan 2001. Internal publication available at `http://hal.inria.fr/inria-00072527/en/`.

[99] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 171–183, New York, NY, USA, 1996. ACM Press.