Chapter 1

Nitro: A low-level functional language

Allan Clark[1]

Abstract: Nitro is to be a small functional language. Setting Nitro apart from the many other functional languages available to programmers will be that Nitro code will be low-level, and more of the underlying machine will be visible to the programmer. This allows functional programmers to write inherently low-level code such as operating system code, device drivers and run times for higher level languages. Programs written in Nitro will be guaranteed to have certain safety properties. Therefore writing low-level components in Nitro will have two benefits, the safety properties of Nitro programs will be guaranteed, and writing in Nitro will be familiar to anyone accustomed to programming with a functional language such as SML[1].

This paper gives a definition of the core of the Nitro programming language.

1.1 INTRODUCTION

The C programming language is ubiquitious in low-level systems programming. It is rare for programs such as operating system kernels, low-level device drivers, and abstract machines to be written in languages other than C. One good reason for this is that C provides the control over the underlying machine that many other high level languages are designed to abstract away from.

The aim of the Nitro programming language is intended to provide to low-level programmers the safety, convenience and clarity that has been afforded to high level functional programmers for many years. We can then be more confident that our low-level code is safe and correct. Given that our high-level code will run on top of this low-level code we obtain the added benefit of improving the confidence we have in our total trusted computing base.

[1]Laboratory for Foundations of Computer Science, The University of Edinburgh, King's Buildings, Edinburgh, EH9 3JZ, Scotland;Email: `A.D.Clark@sms.ed.ac.uk`

In the remainder of this paper I give an initial list of aspects of functional programming that would be desirable to a low-level programmer. I then give an informal description of the abstract syntax of the language, and finally a definition of that abstract syntax in operational semantics.

1.2 THE ABSTRACT SYNTAX

In this section I present the abstract syntax of the core of the Nitro programming language. The core is all of Nitro, minus the modular language and derived forms. This section will take the form of a list of the core Nitro constructs and an informal and brief semantics of each clause. Each clause is presented using an SML like notation, those readers unfamiliar with SML can think of this a mathematical notation. In particular one should keep in mind that each clause is merely a representation of that clause designed to be easily readable and unambiguous. The semantics in this section is merely to give the reader an overview of the language and highlight the interesting parts, however, when accompanied with the syntax of the full language, should be enough to allow the reader to begin programming in Nitro. The full formal semantics will be given in the sections that follow.

1.2.1 Simple Objects

We use the word object here to refer to a data value, something which can be passed around within the language, we make no reference to object oriented programming. We begin with a brief look at the types of objects in Nitro. This is a necessary beginning as we cannot describe the means by which we manipulate objects in Nitro before we have described what it means to be an object in Nitro.

There are five simple types in Nitro:

- int
- float
- boolean
- char
- string

Notice that a string is distinct from a list of characters.

1.2.2 Type constructions

Other types can be constructed in one of these principal ways.

Tagged(constructors) A tagged union type consists of a set of constructors, each of which takes one argument, which may be a complex type, and returns the tagged union type. Tagged union types should be familiar to most functional programmers.

Tuple_type (type list) A tuple type is similar to a structure, except that each component is accessed by position rather than name. Again tuple types should be familiar to most functional programmers.

Fn_type ((string * type) list * type) A function will take in a list of arguments, and return a value. Note that the list of arguments is not a datatype, that is they are values separated from each other. Also within the type is stored the name of the argument if given within the type. This means that if a type constraint is given with a name for a parameter then the implementation must have the same name for that parameter. The logic behind this is a type constraint given in a signature. In a signature we can constrain a value to have a function type, say

```
Fn_type ((("height", int), ("width", int)), int)
```

suppose that this is the type of a function named draw_rect, which is supposed to draw a rectangle on the screen. A user of this interface can then see that the height should be given as the first argument and the width as the second. Without the argument names there the programmer would be forced to look at the implementation, which might not be available. The checking of the type of the implementation would then force the implementor to give the argument names such that height is first, and width second. Of course this does not stop the implementation from drawing a rectangle that is `width` tall, but it is likely to prevent this, and/or make the bug easier to find.

Tscheme (id list * scheme) A type scheme provides a mechanism to define a general complex type that can be parameterised by a number of simple types. The id list is a list of arguments to the type scheme. The type scheme may consist of the other type constructions above, and can make use of the type scheme's parameters. We can then instantiate the parameters to make a new type.

1.2.3 Type Matching

We now describe what it means for a type to match another type. This will be useful when we come to describe the semantics of expressions, and we wish to say, the expression is typeable as long as the type of a sub component matches some other given type. Essentially a type `t1` matches another type `t2` if `t1` is at least as general as `t2`. So for example the built-in type `int` can only be matched by itself, the polymorphic function type `Fn_type (("", 'a), 'a)` can be matched by any function type, that has one parameter and returns a value of the same type as its parameter. Function types have the special meaning where the string components of the parameters must be compatable. The empty string is compatible with any string, while a non-empty string is only compatible with an identical string.

1.2.4 Expressions

We are now ready to begin our look at the semantics of the Nitro language. We begin with expressions which form the main part of the language. An expression may be:

IntConst (n) Where n is an integer constant. The expression is always typeable. The expression has type int.

FloatConst (f) Where f is a floating point constant. The expression is always typeable. The expression has type float.

BoolConst (b) Where b is one of `true` or `false`. The expression is always typeable. The expression has type bool.

StringLiteral s Where s is a literal string given in the text of the program. The expression is always typeable. The expression has type string.

Path (p) Where p is a path to an identifier. Concerning ourselves with only the core of the language for the time being, a path can simply be an identifier. The expression is typeable provided that the identifier referred to by the path is currently in scope. The type of the expression is given by the type of the identifier in the current context.

Apply (expr * expr list) The first expression must evaluate to a value with function type, this function is then applied to the list of expressions. Notice that the function is applied to a list of expressions, where the list of expressions is not curried arguments or a value of tuple type or a value of some type list, but simply a list or sequence of arbitrary expressions. In particular the function expression must be applied to the whole list of argument expressions at the same time. This differs from most functional languages where functions only take one argument, and multiple arguments are simulated by tuple types, and the use of curried functions.

The expression is typable whenever the first expression admits a type of the form $Fn\_type(((name_1, type_1) \ldots (name_n, type_n)) * result\_type)$, and the list of expressions admit types $t_1 \ldots t_n$, and the list $t_1 \ldots t_n$ is compatible with the $type_1 \ldots type_n$. The type of the expression is then $result\_type$.

Object Construction An object construction can be used to build a more complex object from simpler constituent objects. Below is a list of such possible expressions.

Constructor a Where `Constructor` is a tagged union type constructor and `a` is its argument. The expression is typeable, whenever the expression `a` is typeable and admits a type `t` and `Constructor`, in the current context accepts an argument of type `t'` and `t` is compatible with `t'`. The type of the expression is the tagged union type to which `Constructor` belongs according to the current context.

Tuple (expr list) Creates a tuple value with arity equal to the length of the list of expressions. The expression is typeable whenever all of the expressions in the expression list are typeable Where the expression list has types $t_1 \dots t_n$ the whole tuple expression has type, $Tuple\_type(t_1 \dots t_n)$

MatchRules ((p list * expr) list) A set of match rules are not an expression by themeselves but are used in two definitions below so they are included separately here. Each match rule consists of a list of patterns and an accompanying expression. A pattern is a plan of an object against which a value can be matched. A pattern is essentially a construction of a value, using the Object Construction expressions described in the list above. In addition a pattern can contain one or more holes which represent that any object of the correct type can be used to match that part of the pattern. A hole in a pattern is generally an identifier, where this identifier is then bound to the part of the expression being matched and can then be used in the evaluation of the expression accompanying the pattern list in the current match rule. Below is a list desribing the forms a pattern can take, the identifiers bound by that pattern and the string, type pair associated with a type correct pattern of that form.

A type correct pattern, will be given a pair consisting of a string and a type. A match rule is typeable, when all of its patterns are typeable, and the expression is typeable in the context obtained by adding the identifiers bound by the patterns to the current context. A type correct match rule produces a function type which has parameters equal to the type of the patterns and a result type equal to the type of the expression.

A set of match rules, is then typeable, when all of the match rules are typeable and they all have matching types. Recall the meaning of matching function types given in the section Type Matching. The function type assigned to the match rules is equal to the least general function type, which is a match for all of the function types obtained from the list of match rules.

Here is a list of what a pattern can be, along with the identifiers that are bound by it, when the pattern form is typeable and the string, type pair associated with a type correct pattern of that form.

Identifier id
- Expressions - Matches all expressions
- Identifiers - `id` is bound to the value of the matched expression
- Type result - `("", 'a)`

Wildcard
- Expressions - Matches all expressions
- Identifiers - No additional identifiers are bound
- Type result - `("", 'a)`

Intconst n
- Expressions - An integer expression with the value `n`
- Identifiers - No additional identifiers are bound
- Type result - `("", int)`

Boolconst b
- Expressions - A boolean expression with the value `b`

- Identifiers - No additional identifiers are bound
- Type result - `("", bool)`

UnitExp
- Expressions - A unit expression
- Identifiers - No additional identifiers are bound
- Type result - `("", unit)` Unit patterns are not really useful, since they could always be matched with _, however it does force the type of the expression matched to be `unit` which can prevent a bug such as a partial application of a curried function.

Tuple (pattern list)
- Expressions - Matches any tuple value that has arity equal to the length of the given pattern list, and whose component values match the given patterns.
- Identifiers - No additional identifiers are bound, but the patterns in the given pattern list may bind identifiers.
- Type result - If the pattern list gives types $(t_1 \ldots t_n)$ then the result is `("", Tuple_type(`$t_1 \ldots t_n$`))`

Constructor (pattern)
- Expressions - A value which has been constructed using the given constructor, and whose argument matches the pattern `pattern`
- Identifiers - No additional identifiers are bound, but there may be identifiers bound by `pattern`
- Type result - Where `t'`$\rightarrow$ `t` is the type of the given Constructor in the current context, and `t'` matches the tpe of the pattern `pattern` then, `("", t)`

As (p, id)
- Expressions - Matches any expression which matches the pattern `p`
- Identifiers - The identifier id is bound to the matched expression
- Type result - Where `p` has type `t`, and the name of the identifier is `s`, `(s, t)`

Match expr list * matchrules We evaluate the given expression list, and then each match rule is matched against in the order in which they are written. The first match rule whose pattern list matches the values of the expression list has its associated expression evaluated to give the result of the the whole match expression. The expression is typeable, where `matchrules` are typeable and evaluate to the function type `Fn_types((`$(s_1,t_1)\ldots(s_n,t_n)$`), result_type)`. The given expression must be typeable and have types `(`$t'_1 \ldots t'_n$`)` and the types `(`$t'_1 \ldots t'_n$`)` must match the types `(`$t_1 \ldots t_n$`)` The type of the whole expression is then `result_type`, though this may be further constrained by the type of the expressions.

Fn (matchrules) As in most functional languages Nitro provides a way to write down a function as an expression. The major difference is that the matchrules here can have a list of patterns that match a list of expressions, where the expressions are, unlike within a tuple expression, separate expressions. This decision allows us to interact with most low-level language functions, notably C functions, in a natural manner. In addition allowing the programmer to differentiate between mutiple values and tuple values, allows the programmer to help the compiler produce efficient code, see [2].

The expression is typeable when `matchrules` are typeable and omit the function type `fun_type` which is then the type of the whole expression.

Embed (pattern, code, expr) One of the goals of Nitro is to allow the interaction with existing low-level languages. We do this by allowing interaction with the host language. The host language is which ever language the implementation in use is compiled to. For example this paper describes an implmentation that compiles Nitro to Cyclone code, hence using that implementation a programmer is able to embed Cyclone code. For typing purposes embed is seen as an expression. It is upto the programmer to ensure that this expression is used with a safe type. The given `pattern` is used to declare variables, which must be set by the embedded code. In general the code will use side effecting assignments to achieve this. To what extent the code can use identifiers not within the pattern, but currently in scope, is left undefined, and is implementation specific. For the Cyclone implmentation, `code` is a cyclone statement, usually used to set one or more of the identifiers declared within `pattern`. The `expr` will then use these identifiers to build up a value, commonly either a single identifier or a tuple expression of identifiers. Note that using an embed expression is therefore unsafe, since we make no check on the underlying code, though the implementation is of course free to make as many checks as it deems worthwhile. The expression is typeable, whenever the `pattern` is typeable in the current context, and `expr` is typeable in the context where each identifier declared in `pattern` is added to the enivironment. The type of the whole expression is then the type of `expr`

Reduce (id list, expr) A reduce expression, is similar to a function, however it is expanded inline and the arguments to which it is applied are not evaluated, but are substituted into their equivalent place in the `expr` of the reduce clause. A reduce clause is therefore somewhat similar to a λ abstraction in the λ calculus. However it is in general used here to increase efficiency and therefore is akin to a macro in C. A reduce expression is typeable whenever the expression `expr` is typeable within the context that includes the `id list` into the current context with unknown type. If the types of the identifiers are inferred to be $(\tau_1 \dots \tau_n)$ and their textual representations are $(s_1 \dots s_n)$ and the `expr` has type τ then the whole expression has type $Function((s_1, \tau_1) \dots (s_n, \tau_n) \rightarrow \tau)$. Notice that we can use identifiers that are currently in scope, since the reduce abstraction can only be applied wherever it is in scope, the identifiers that it uses must also still be in scope.

1.3 THE NITRO LANGUAGE

This section aims to give a formal definition of the Nitro programming language. The section Abstract Syntax gave an informal semantics for the abstract syntax of the language, while the section Concrete Syntax, gave the physical forms that the abstract syntax is written in, plus the derived forms provided to allow programs to be written concisely. We begin this section with the static semantics of the abstract syntax. This is followed by a dynamic semantics of the abstract syntax.

1.3.1 Static Semantics

Contexts

We represent our contexts with a C, additional contexts have a prime appended, so we can have the contexts C, C' and C". A context is made up of an environment E, a constructor environment CE and a type environment TE. An environment E is a list of pairs of identifiers and types. The expression

$$E(v) = \tau$$

means that looking through the list of pairs, the first occurence of the identifier v is paired with the type τ. To add to an environment we use

$$E + (v, ty)$$

which means the environment obtained by adding the binding

$$(v, ty)$$

to the environment

$$E$$

A constructor environment CE is a list of pairs of identifiers and pairs of types. The two types associated with an identifier are the argument to the constructor and the resulting tagged union type obtained by applying the constructor.

We use $C(E)$ to mean the environment component of the context C. We can also update a context with a new component, thus

$$C \oplus E'$$

is the context C with the environment component E' in place of E. The type environment consists of a list of type variables that are currently in context and a list of type functions, where a type function is a name and a list of type variables that it takes as arguments. To add a list of type variables to a type environment we use TE + tyvars, and to add a type function we use TE + tyfun.

Before we proceed, we first explain the Closure and MGST operations. Typically we will want to take the Closure of a type with respect to a context. So the closure of a type t with respect to the object O is Closure (O, t), which gives us back a type scheme or type function being tyfun(tyvars, t) where tyvars is all the type variables that occur in t, and do not occur in the object O.

The MGST operation is performed on two types or type schemes and we obtain the most general type of which both the original types are sub types. When we say t1 is a subtype of t2 then we mean that any object that is of type t2 is also of type t1. Any object which is of type `Tuple(bool,bool)` is also of type `Tuple('a,bool)`. For example suppose we take the MGST of the type `Tuple('a, int)` with `Tuple(bool, 'a)` then we obtain the type `Tuple(bool,int)`. Note that within the definition of the MGST there is an implicit check that the types do have a most general super type. For example MGST (int, bool) will fail.

1.3.2 Declarations

We now begin by looking at declarations, we start with type declarations.

Type declarations

First we define the rules for Constructor definitions, here the rules need to contain a result type, that is the type that the constructor will produce.

1.

$$\frac{C \vdash ty \rightarrow \tau'}{C, \tau \vdash Cname\ ty \rightarrow (Cname, (\tau' \rightarrow \tau))}$$

A list of constructors will all produce the same result type, and all add to the constructor environment. We also infer a type, which is just the same as the type passed in along with the context, the reason for this is to keep these rules in the same structure as those rules below for tuple and function types. So where

$$\forall x, 0 < x \leq n\ C, \tau \vdash Cname_x\ ty_x \rightarrow cy_x$$

2.

$$\frac{\forall x, 0 < x \leq n\ C_x = C_{x-1} \oplus (C_{x-1}(CE) + cy_x)}{C_0, \tau \vdash (Cname_1, ty_1 \ldots Cname_n, ty_n) \rightarrow \tau, C_n}$$

Now we add rules to type check ordinary types, tuple and function types, these rules must also produce a Constructor Environment though of course it is just the same one that was used to type check the respect types. Also the conclusion requires both a context and a type, we have ignored the type here.

3.

$$\frac{\forall x, 0 < x \leq n\ C, \_ \vdash ty_x \rightarrow \tau_x}{C \vdash Tuple(ty_1 \ldots ty_n) \rightarrow Tuple(\tau_1 \ldots \tau_n), C(CE)}$$

4.

$$\frac{\forall x, 0 < x \leq n\ C \vdash ty_x \rightarrow \tau_x \quad C \vdash ty \rightarrow \tau' \quad \tau = Function((s_1, \tau_1) \ldots (s_n, \tau_n) \rightarrow \tau')}{C, \_ \vdash Function((s_1, ty_1) \ldots (s_n, ty_n) \rightarrow ty) \rightarrow \tau, C(CE)}$$

Now we deal with the case where the type is a type function. That is we have some type parameters, we deal separately with tagged union definitions which contain constructor definitions. We first must had those to the environment before we type check the body of the type, which may include those type parameters. We use $\{\}$ to mean the empty type scheme, this means that the type is just the type name. So where we have

$$\tau_{var} = tyfun(tyvars, name, \{\})$$

then we have

5.

$$\frac{TE' = TE + \tau_{var} \quad C \oplus (TE' + tyvars), \tau_{var} \vdash ty \rightarrow CE', \tau \quad Closure(\tau_{var}, \tau)}{C \vdash Type\ tyvars\ name = ty \rightarrow C \oplus CE', TE'}$$

notice that we place the type into the environment in which we check the body of the type. This is to allow for recursive type definitions.

Finally we define the application of type function. Here we use the notation tyvars $\rightarrow$ tys to mean that the list of type variables are instantiated to be the given types. We use this to give a type environment in which to take the Closure of the type function given by the tyname. So when

$$TE' = TE + (tyvars \rightarrow (\tau_1 \ldots \tau_n))$$

we have

6.

$$\frac{\forall x, 0 < x \leq n\, C \vdash ty_x \rightarrow \tau_x \quad tyfun(tyvars, name, \tau') = TE(tyname) \quad \tau = Closure(TE', \tau')}{C \vdash (ty_1 \ldots ty_n) tyname \rightarrow \tau}$$

Value declarations

Value declarations are quite straightforward, we simply introduce a new value into the current environment.

7.

$$\frac{C \vdash exp \rightarrow \tau \quad VE' = VE + (name, \tau)}{C \vdash Valuename = exp \rightarrow C \oplus VE'}$$

1.3.3 Expressions

In this section the inference rules have conclusions of the form Environment $\vdash$ expr : type in general either the type is a specific type such as int, or it is a type variable t. In the cases where it is a type variable t, it is most often the case that the sub goals will refer in some way to the type variable t to restrict it to the type of the expression expr according to the environment. In the few cases that this is not true, then it means that the expression can have any type.

Simple Expressions

We begin our look at expressions with the simple expressions. All of these expressions do not contain any sub expressions, therefore their type is dependent only on the expression and the current environment.

8.
$$\frac{}{C \vdash n : int}$$

9.
$$\frac{}{C \vdash f : float}$$

10.
$$\frac{}{C \vdash b : bool}$$

11.
$$\frac{}{C \vdash sl : string}$$

12.
$$\frac{E(v) = \tau}{C \vdash v : \tau}$$

Object Construction

This sections details the expressions that represent constructing an object from simpler objects. We start with the explicit constructors for tagged union data types. Notice that they are similar to an Apply expression, except that a constructor only has one argument, which of course may be a tuple.

13.
$$\frac{C(CE)(Constructor) = \tau' \rightarrow \tau \quad C \vdash arg : \tau'}{C \vdash Constructor(arg) : \tau}$$

Tuple construction forms a single object from many objects.

14.
$$\frac{C \vdash (e_0 \ldots e_n) : (\tau_0 \ldots \tau_n)}{C \vdash tuple(e_0 \ldots e_n) : \tau} \; \tau = Tuple\_type(\tau_0 \ldots \tau_n)$$

Matching

In this section we describe the matching rules. We begin with patterns, the rules for patterns have a conclusion of the form

$$C \vdash pattern \longrightarrow (s, ty, E)$$

where s is the string associated with the pattern, ty is the type associated with the pattern and E is a new environment obtained by adding the identifiers introduced by the pattern to the current context's environment. In these rules the value name(id) correspondes to the textual representation of the identifier id.

15.
$$\frac{}{C \vdash Identifier(id) \longrightarrow (name(id), \tau, C(E) + (id, \tau))}$$

16.
$$\frac{}{C \vdash WildCard \rightarrow ("", \tau, C(E))}$$

17.
$$\frac{}{C \vdash IntConst(n) \rightarrow ("", Int, C(E))}$$

18.
$$\frac{}{C \vdash BoolConst(b) \rightarrow ("", Bool, C(E))}$$

19.
$$\frac{}{C \vdash UnitExp \rightarrow ("", Unit, C(E))}$$

A list of patterns is type checked in the intuitive way. We return a list of strings and types, the strings are disregarded when type checking a tuple pattern, however they are required when type checking a match rule. 20.

$$\frac{}{C \vdash \varepsilon \rightarrow (\varepsilon, E)}$$

21.

$$\frac{C \vdash p \rightarrow (s, ty, E'') \quad C \oplus E'' \vdash patterns(tys, E')}{C \vdash p :: patterns \rightarrow ((s, ty) :: tys, E')} \; where\ no\ identifer\ occurs\ twice$$

22.
$$\frac{C \vdash patterns \rightarrow (((s_1, t_1) \ldots (s_n, t_n)), E')}{C \vdash Tuple(patterns) \rightarrow ("", Tuple(t_1 \ldots t_n), E')}$$

23.

$$\frac{C(CE) \vdash Constructor \rightarrow (\tau' \rightarrow \tau) \quad C \vdash pattern \rightarrow (s, \tau', E')}{C \vdash Constructor(pattern) \rightarrow ("", \tau, E')}$$

24.

$$\frac{C \vdash p \rightarrow (s, \tau, E')}{C \vdash As(id, p) \rightarrow (name(id), \tau, E' + (id, \tau))} \; id\ does\ not\ occur\ in\ p$$

So now a match rule consists of a list of patterns and an associated expression. We first have to type check the patterns and then the expression in the environment given by typechecking the pattern list

25.
$$\frac{C \vdash patterns \rightarrow (stys, E') \quad C \oplus E' \vdash exp \rightarrow \tau}{C \vdash (patterns, exp) \rightarrow (stys \rightarrow \tau)}$$

And a list of match rules must have compatible types

26.
$$\frac{}{C \vdash \varepsilon \rightarrow \tau}$$

27.

$$\frac{C \vdash mrule \rightarrow \tau' \quad C \vdash mrules \rightarrow \tau'' \quad C \vdash MGST(\tau', \tau'') \rightarrow \tau}{C \vdash mrule :: mrules \rightarrow \tau}$$

Compound Expressions

Now we define the types of expressions that contain other expressions. We begin with application.

28.

$$\frac{C \vdash e : ((s_0,\tau_0)\ldots(s_n,\tau_n)) \to \tau \quad C \vdash (e_0 \ldots e_n) : (\tau_0 \ldots \tau_n)}{C \vdash Apply(e,(e_0 \ldots e_n)) : \tau}$$

29.

$$\frac{C \vdash mrules \to \tau}{C \vdash Function(mrules) \to \tau}$$

To type check a match expression is effectively the same as an application of a function expression.

30.

$$\frac{C \vdash exprs \to (\tau_1 \ldots \tau_n) \quad C \vdash mrules \to ((s_1,\tau_1')\ldots(s_n,\tau_n') \to \tau)}{C \vdash Match(exprs, mrules) \to \tau}$$

The embed expression is type checked basically without type checking the embedded code. We rely on the embedded code being type safe. This means that an embed expression is type checked just as a match rule.

$$\frac{C \vdash pattern \to (s,\tau',E') \quad C \oplus E' \vdash exp \to \tau}{C \vdash Embed(pattern, code, expr) \to \tau}$$

Similarly a reduce expression is type checked almost like a function. The difference being that there is only one match rule, and each pattern in the match rule must be an identifier. So for each identifier we give it a new type variable and then add the pair to the current environment. This gives us an environment in which to type check the expression which gives us the type of the whole expression.

$$\frac{\forall 0 < x < n \vdash id_x \to \tau_x \quad \forall 0 < x < n \vdash C_x = C_{x-1} \oplus (id_x,\tau_x) \quad C_n \vdash expr \to \tau}{C \vdash Reduce(id_1 \ldots id_n, expr) \to \tau} \; C_0 = C$$

1.3.4 Dynamic Semantics

We now look at the dynamic semantics of the core of Nitro.

Simple Expressions

We begin with simple expressions, the rules in this section all have the form

$$C \vdash exp \to v$$

1.

$$\frac{}{C \vdash n \to n}$$

2.
$$\frac{}{C \vdash f \rightarrow f}$$

3.
$$\frac{}{C \vdash b \rightarrow b}$$

4.
$$\frac{E(id) = v}{C \vdash id \rightarrow v}$$

Matches

A match is a set of match rules. We may fail, which means that we do not match any of the match rules. The first rule is simple, if there are no match rules then we fail automatically.

5.
$$\frac{}{E, vs \vdash \varepsilon \rightarrow FAIL}$$

After that if there are still match rules to try then we attempt to match with the first match rule, if so then we evaluate the associated expression, else we try to match on the rest of the match rules.

6.
$$\frac{E, vs \vdash mrule \rightarrow v'}{E, v \vdash mrule :: mrules \rightarrow v'}$$

7.
$$\frac{E, vs \vdash mrule \rightarrow FAIL \quad E, vs \vdash mrules \rightarrow v'}{E, v \vdash mrule :: mrules \rightarrow v'}$$

So a match rule is successfully matched if each pattern matches the corresponding value. The result of the whole match rules is then the result of the expression of the match rule given that all of the identifiers that are bound in the set of patterns are added to the value environment.

8.
$$\frac{\forall x, 0 < x \leq n \; E_{x-1}, v_x \vdash pat_x \rightarrow E_x \quad E_n \vdash exp \rightarrow v'}{E, v_1 \dots v_n \vdash (pat_1 \dots pat_n \rightarrow exp) \rightarrow v'} \; E_0 = E$$

We are therefore left to define what it means for a pattern to be matched by a value, and the resulting new value environment. A wildcard matches any expression, and adds nothing to the value environment.

9.
$$\frac{}{E, v \vdash \_ \rightarrow E(VE)}$$

Similarly a constant pattern matches the exact value it denotes and again adds nothing to the value environment.

10.
$$\frac{v = con}{E, v \vdash con \rightarrow E(VE)}$$

We may also fail a pattern match against a constant

11.
$$\frac{v \neq con}{E, v \vdash con \rightarrow FAIL}$$

1.4 CONCLUSION

The Nitro programming language hopes to remedy the lack of functional programming languages available to the systems programmer. The convenience and safety of functional programming languages is without question. The efficiency is sometimes called in to question, however more and more researchers are beginning to believe that with a smart compiler a functional language need be no less efficient than an imperative one. Furthermore research in garbage collection is advancing to the point where for some applications garbage collection can be more efficient than manual memory management. Everyone in favour of functional programming languages will argue that such a programming language allows us to write, correct and secure programs by remaining type safe and catching common programmer mistakes at compilation time. However these techniques have yet to be applied successfully in the one area where one could argue that these features are most required, that of low-level systems programming. Where we require that our programs are secure and robust and we should not tolerate frequent program errors and failures.

REFERENCES

[1] R. MILNER, M. TOFTE, R. HARPER AND D. MACQUEEN (1997), "The Definition of Standard ML (revised)", MIT Press, Cambridge

[2] K. MITCHELL (1994), "Multiple Values in Standard ML", LFCS, University of Edinburgh

[3] T. JIM, G. MORRISETT, D. GROSSMAN, M. HICKS, J. CHENEY AND Y. WANG (2002) "Cyclone: A Safe Dialect of C" USENIX Annual Technical Conference, pages 275–288, Monterey, CA.

[4] B.J. MCADAM (2002) "Repairing Type Errors in Functional Programs" LFCS, University of Edinburgh

[5] L. CARDELLI (1989) "Typeful Programming" Digital Equipment Corporation, Systems Research Center

[6] E. ANDERSEN "Busybox" http://www.busybox.net/

[7] J.M.BELL, F. BELLEGARDE AND J. HOOK "Type-driven Defunctionalisation" Pacific Software Research Center