FAST compiler user's guide

Pieter Hartel * Hugh Glaser † John Wild ‡

Abstract

The FAST compiler is a backend for compilers of lazy functional languages. There are two versions of the compiler: one that takes a rather simple lazy functional language as input and a second that accepts a language similar to Miranda.

On output the compiler produces a set of macro calls that are normally turned into a C program by one of the code generators that have been developed for FAST. Such a C program must be compiled by a C compiler and linked with the appropriate runtime library to form an executable.

This document decsribes how to use the FAST compiler. Familiarity with functional languages and their implementation methods is required to make full use of this document.

1 Introduction

The FAST (Functional programming on ArrayS of Transputers) project team at Southampton has developed an optimising compiler for a lazy functional language on a single processor [6]. This document describes how to use the compiler. The compiler accepts on input a language called *Intermediate* that is best described as a mixture of Miranda[1] [17, 18] Haskell [14] and LML [1, 2]. The language was originally not intended to be used as a programming language but as the target of a frontend compiler. However over the years Intermediate has been extended to a point where the name has become less appropriate.

On output the compiler produces a set of macro calls that are normally turned into a C program by one of the code generators that have been developed for FAST. Such a C program must be compiled by a C compiler and linked with the appropriate runtime library to form an executable.

There are two versions of Intermediate and also of the FAST compiler, each with a different purpose and use:

Basic Intermediate is used as the target language for the Haskell- frontend developed at Imperial College. This frontend compiles a subset of Haskell into basic Intermediate.

Advanced Intermediate has several more high level features (algebraic data types, list comprehensions and arithmetic sequences) so that it can be used without a frontend. To use advanced Intermediate effectively a separate program development system (for which we use the Miranda system) is required. Here the role of the frontend has been reduced to a number of textual changes that are normally performed by a simple sed

*Computer Systems Department, University of Amsterdam
†Department of Electronics and Computer Science, University of Southampton
‡XAM Ltd, Armley, Leeds, England
[1] Miranda is a trademark of Research software Ltd.

script, so that in principle the same program is acceptable by both the advanced Intermediate compiler and the regular program development system. This approach has been chosen based on the observation that program development and debugging is sufficiently different from the generation of efficient code that it warrants the use of completely separate tools. The FAST compiler may devote all its attention to the generation of good code and need not bother with for example printing meaningful error messages at compilation and at runtime. This does not imply that a development system could not generate good code, but simply that separating the two concerns makes life a lot easier.

The features that are lacking in both forms of Intermediate are separate compilation, operator overloading and polymorphic equality. There is also a restriction on the possible forms that local function definitions may take (see section 3). Neither version of the FAST compiler performs type checking although both perform type inference along with a host of other program analysis and optimisations for code generation purposes.

There are also different code generator and associated runtime systems that can be used with the two FAST compilers:

Documentation code generator and runtime were designed to aid in the documentation of the compiler [7]. At present the documentation runtime system does not have a garbage collector.

Production code generator and runtime were primarily built to gather extensive statistics [21]. This system has a reference counting garbage collector.

Performance code generator and runtime were built (by Koen Langendoen of Amsterdam University) to achieve best performance [11]. This system has a two-space copying garbage collector.

Figure 1 gives an overview of the software that we use for developing functional programs. The Haskell- route and the Miranda route should be more integrated than they are at present. It is possible to generate Haskell, Concurrent Clean [13, 19] and LML from advanced Intermediate, so that implementations can be compared using the same benchmark programs [8].

The recommended way to use advanced Intermediate is as follows:

1. Develop the program using the Miranda system, freely using the primitives provided by Intermediate (e.g. array support) and at the same time taking into account the limitations imposed by Intermediate (e.g. the lack of support for operator overloading).

2. When the development is complete, use the FAST compiler to generate an executable image for the architecture of choice. This could be a parallel system supporting one or more of the primitives that have been implemented in the FAST runtime systems.

Various parallel systems are being built using the FAST compiler as a code generator for the sequential processing elements. The group of Paul Kelly and Stuart Cox at Imperial College London has embedded the FAST compiler in the Caliban system [10], which enables the programmer to annotate programs so that they may be run in parallel on a network of transputers [5]. The Imperial College team are using the production code generator and runtime. Koen Langendoen has built a system based on shared memory clusters (the HyperM system [12]) that allows divide and conquer parallelism to be exploited. This system uses the

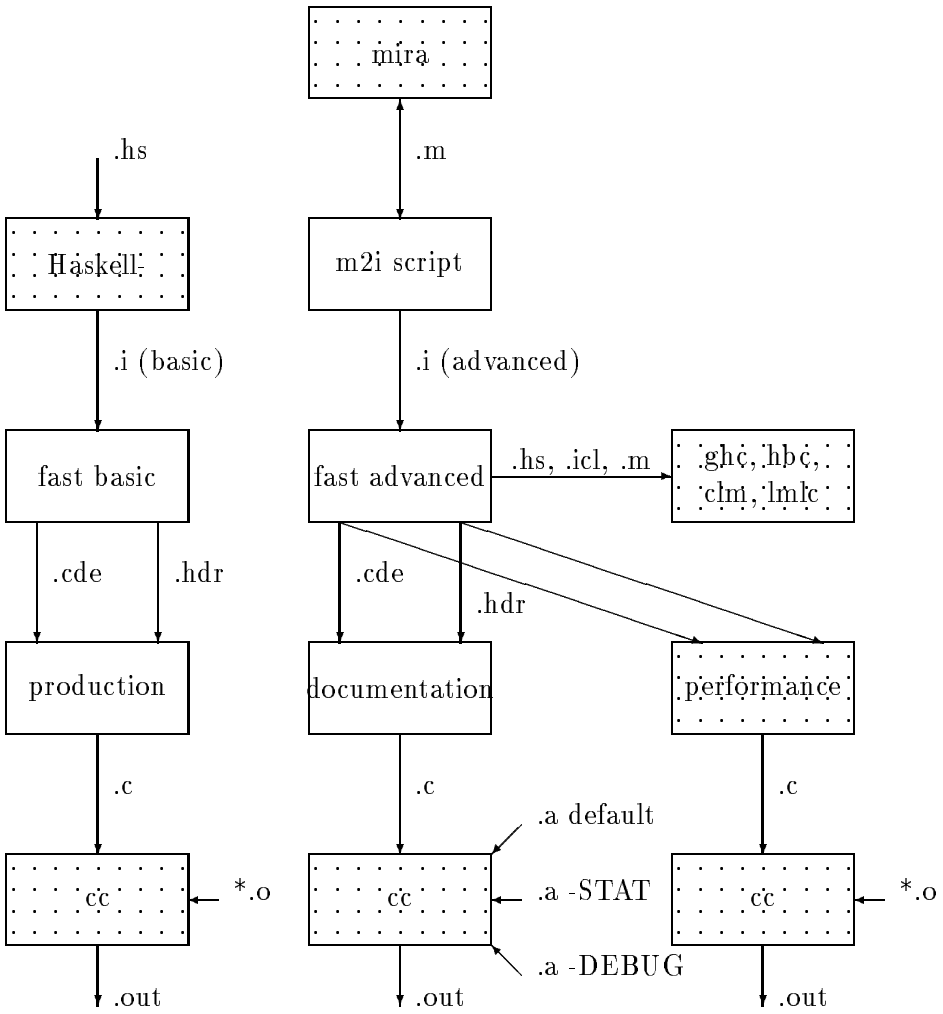


Figure 1: Structure of the FAST system with work by others shown in dotted boxes. File name extensions are shown along the arrows.

performance code generator and runtime. The rest of the document describes how to use the compiler to generate code for a sequential machine. The description often refers to the Miranda System Manual and Haskell Language manual. This document is primarily intended to describe advanced Intermediate since basic Intermediate is not normally seen by human eyes. The syntax of basic Intermediate will be given nevertheless.

We apologise for not having implemented all the standard functions described here in every runtime system. In particular some of the array primitives will be missing, and each runtime system will typically implement one of the functions for parallelism.

The syntax and semantics of basic and advanced Intermediate are described in Section 2. Some important restrictions on the use of advanced Intermediate are discussed in Section 3. The options and flags to the FAST compiler are described in Section 4. Section 5 describes how to use the documentation runtime system. Sections 6 and 7 describe the production runtime system. Concluding remarks may be found in Section 8.

2 Syntax and semantics

The lexical structure of the language describes the elements that are used in the syntax description that follows.

2.1 Lexical structure

Identifiers consist of letters, digits, underscores and single quotes. Identifiers that begin with an upper case letter are taken to be constructor names (which must be declared in an algebraic data type). Ordinary identifiers must begin with a lower case letter. Type identifiers are `*`, `**`, `***` etc.

An integer is a sequence of digits (e.g. `42`). A floating point number consists of a (non empty) mantissa and an optional exponent, which may be signed. A floating point number must either have a decimal point or an exponent to distinguish it from an integer (e.g. `4.2` or `1e2`). There is no concept of a negative numeric constant. For example `-42` is interpreted as an expression that applies the unary negation function `-` to the integer `42`.

There is no automatic coercion of integers to floats, so that you must write `1.0/2.0` to make sure that the floating point divide operator receives two floating point operands. There is no support for arbitrary precision integers; integers and floating point numbers are at least 31 bits, depending on the code generator/runtime system being used. The present status of the implementations is:

runtime	single precision float	double precision float
Documentation	not supported	implemented as 64 bit double
Production	not supported	implemented as 32 bit float
Performance	implemented as 31 bit float	implemented as 31 bit float

Character constants are enclosed in single quotes as in C (e.g. 'a'). The following escape sequences are recognised: `'\n' '\t' '\f' '\r' '\b' '\\' '\''`. String constants are enclosed in double quotes, also as in C (e.g. `"foo\n"`). String constants may not extend beyond one line. Comments are introduced by two adjacent vertical bars `||` and extend to the end of the line. White space (tab, form feed, space and newline) may be inserted freely. There is no offside rule as in Miranda; instead semicolons must be written to terminate definitions.

2.2 Standard identifiers

There are a few keywords and many standard identifiers in Intermediate. These may not be redefined or used as local variables. (This is different in Miranda).

2.2.1 Proper keywords

The keywords are listed here with a short explanation of their use. Refer to the syntax for more information. (Miranda does not support `INLINE` or `LET ... IN`).

keyword	description
`INLINE`	Used to inline a function definition
`LET IN`	Used in let expressions
`if otherwise`	Used in guarded function definitions
`where`	Used in where expressions
`abstype with`	Ignored but accepted for compatibility with Miranda

2.2.2 Special symbols

Special symbols when made up of several characters must be spelled as is, i.e. without intervening layout. Refer to the syntax for more information.

symbol	description
`==`	Used in defining type synonyms
`::`	Used in specifying a type signature
`::=`	Used in defining algebraic data types
`|`	Announces an alternative in an algebraic data type
`->`	Function type constructor
`;`	Terminates definitions
`,`	Separates elements of lists and tuples
`( )`	Group expressions
`[ ]`	Used in constructing lists and in list comprehensions
`<-`	Used in list comprehensions
`..`	Used in arithmetic sequences

2.2.3 Built-in types

The compiler knows about a few built-in data types, so that it is easier to efficiently implement functions operating on these data types. An array is implemented as a contiguous block of store, so that indexing can be done in unit time. The array descriptor is a tuple that is strict in its components, unlike ordinary tuples, which are lazy in all components. Associations are strict in the first component but not in the second. Complex numbers have two strict components, also to improve efficiency. Arrays and complex numbers are not part of Miranda.

There are no automatic coercions of one numeric type into another, instead standard functions are provided to go from one representation to another. Miranda provides the necessary coercions automatically.

identifier	type	description
`array_type`		A linear array of any element type
`assoc_type`	== (int,*)	Association used in the construction of arrays
`descr_type`	== (int,int)	Descriptor used in the construction of arrays
`bool`	::= True — False	
`char`		Ascii characters
`complex_type`	== (double,double)	Complex numbers
`double`		Double precision floats
`float`		Single precision floats
`int`		Integer numbers

2.2.4 Unary operators

There are only three unary operators.

operator	type	description
`-`	:: int→int	Unary minus
`~`	:: bool→bool	Logical negation
`#`	:: [*]→int	List length

2.2.5 Binary operators

Most binary operators are defined on integer arguments. The six comparison operators are defined only on integer arguments. Standard functions are available for comparisons on other numeric data types. This is a major difference between Intermediate and Miranda. It is also the source of difficulty when converting Miranda programs to Intermediate, because comparisons, and in particular the equality test, are often used on arbitrary data structures. Such tests must be programmed explicitly in Intermediate. Consider the Miranda program below, which sorts a list of "tree" data as an example:

```
tree           ::= Leaf | Branch num

poly_sort :: [*] -> [*]
poly_sort []     = []
poly_sort (x:xs) = poly_sort [u | u <- xs; u < x] ++
                   [x] ++    [u | u <- xs; u = x] ++
                   poly_sort [u | u <- xs; u > x]
```

To translate this program into Intermediate requires all the comparisons to be explicitly written in terms of primitive comparisons. Here is the result (also note the use of semi colons to terminate definitions):

```
tree           ::= Leaf | Branch num

less             = -1;                        || see text
```

```
equal           = 0;
greater         = 1;

cmp_tree :: tree -> tree -> int;
cmp_tree (Leaf)     (Branch y) = less;
cmp_tree (Leaf)     (Leaf)     = equal;
cmp_tree (Branch x) (Branch y) = x - y;      || see text
cmp_tree (Branch x) (Leaf)     = greater;

tree_sort :: [tree] -> [tree];
tree_sort []     = [];
tree_sort (x:xs) = tree_sort [u | u <- xs; cmp_tree u x < equal] ++
                   [x] ++    [u | u <- xs; cmp_tree u x = equal] ++
                   tree_sort [u | u <- xs; cmp_tree u x > equal];
```

The reason why explicit numeric values were used rather than an algebraic data type is so that the comparison in the third clause of the `cmp_test` function can be written as `x - y`.

In this example the sort function is specific for the "tree" data. It could have been made completely general by supplying an extra argument that takes care of the comparisons.

The binary operators available in Intermediate are:

operator	type	description
`++`	:: [*]→[*]→[*]	Polymorphic concatenation of lists
`--`	:: [int]→[int]→[int]	Subtraction of lists of integers. e.g. `[1,3,3,2,2] -- [3,2,3] = [1,2]`. Note the difference between the type of this operator and the Miranda `--` operator, which is polymorphic.
`:`	:: *→[*]→[*]	Prepend (cons) element to list
`\/ &`	:: bool→bool→bool	Logical operators with short circuit semantics
`> >= = ~= <= <`	:: int→int→int	Integer comparisons, in Miranda these operators are polymorphic
`+ * div mod -`	:: int→int→int	Integer arithmetic
`/ ^`	:: double→double→double	Double precision division and exponentiation operators
`!`	:: [*]→int→*	List indexing, starting at 0
`.`	:: (**→***)→(*→**)→*→***	Function composition
`$`		Syntactic operator to turn a function or constructor name into an infix operator (see section 3.4 for an example of use)

2.2.6 Standard functions

There are a number of miscellaneous functions. One of these (`compose`) is identical to the built-in dot operator. This is done so that with each operator there is an explicit name

associated that can be used in the compiler output. The type specification has been omitted for these "redundant" functions, in the table below and in subsequent tables, as the type of the function is the same as that of the operator.

identifier	type	description
`abort`	:: [char]→*	Abort program with message
`compose`		`(.)` i.e. `compose` is identical to the section of the dot operator `.`
`converse`	:: (*→**→***)→**→*→***	`f x y = f y x`
`delay`	:: *→*	reserved
`fix`	:: (*→*)→*	`fix f = f (fix f)` (see section 3)
`force`	:: *→*	Evaluate argument to full normal form, then return it
`iff`	:: bool→*→*→*	The ordinary three way conditional function
`iffrev`	:: *→*→bool→*	Same as `iff` but with the arguments reversed. This function is sometimes generated by the compiler itself
`seq`	:: *→**→**	Evaluate first argument to head normal form (HNF) then return second

2.2.7 Type testing functions

The FAST compiler uses first order polymorphic type inference to enable the code generator to optimise towards the data types used. However, it is possible, and sometimes necessary to write functions that cannot be type checked by a Hindley-Milner style type checker. To enable such functions to be written, which include for example `show` or generic functions to harness parallelism, a number of runtime type testing functions are being provided. Incorrectly typed functions cannot be written in Miranda, and they should not be written in Intermediate unless necessary in cases such as discussed above.

identifier	type	True if:
`isbool`	:: *→bool	Argument represents a boolean
`ischar`	:: *→bool	Argument represents a character
`islist`	:: *→bool	Argument represents a list
`isnum_d`	:: *→bool	Argument represents a double
`isnum_f`	:: *→bool	Argument represents a float
`isnum_i`	:: *→bool	Argument represents an int
`isnum_x`	:: *→bool	Argument represents a complex

2.2.8 List processing functions

A number of list processing functions are built in to provide more scope for optimisation in the runtime system. For most of the functions below there is a more pleasant syntax available

(e.g. arithmetic sequences and infix `++`). This syntactic sugar is translated into the functions listed below at some stage of the compilation process, so there is no performance penalty in the use of the syntactic sugar.

identifier	type	description
`append`		`(++)`
`cons`		`(:)`
`from_i`	:: int→[int]	`from_i x=[x..]`
`fromthen_i`	:: int→int→[int]	`fromthen_i x y=[x,y..]`
`fromthento_i`	:: int→int→int→[int]	`fromthento_i x y z=[x,y..z]`
`fromto_i`	:: int→int→[int]	`fromto_i x z=[x..z]`
`hd`	:: [*]→*	`hd (h:t)=h`
`tl`	:: [*]→[*]	`tl (h:t)=t`
`length`		`#`
`null`	:: [*]→bool	`null []  =True;` `null (x:xs)=False;`
`pair`	:: [*]→bool	`pair []  =False;` `pair (x:xs)=True;`
`project`		`(!)`
`remove`		`(--)`
`strcmp`	:: [char]→[char]→bool	Compare the argument strings for equality

A function such as `from_i` is actually a member of a family of functions, one for each possible type of number that is supported. The suffix used tells the compiler what argument and result types to expect:

suffix	type
`_d`	:: double
`_f`	:: float
`_i`	:: int
`_x`	:: complex_type

This list could be extended in future should the need arise for example to add quad numbers (`_q`) rational numbers (`_r`) or arbitrary precision integers (`_a`). Not all arithmetic functions are available at the moment, but their names are reserved nevertheless. I.e. `sin_i` cannot be redefined anywhere in an Intermediate program (it draws the error message "reserved identifier"). In Miranda these identifiers may be used freely. This aspect requires some attention when converting Miranda programs into advanced Intermediate.

2.2.9 Integer functions

For most arithmetic functions there is an infix operator available. The majority of the operators expect integer numbers as arguments (exceptions are `/` and `^`). In Miranda the arithmetic operators are applicable to all kinds of numeric values.

identifier	type	description
`add_i`		`(+)`
`code`	:: char→int	Return the ASCII code of the character argument
`decode`	:: int→char	Return the character corresponding to the given ASCII code
`div_i`		`(div)`
`i_to_f`	:: int→float	convert integer to float
`i_to_d`	:: int→double	convert integer to double
`mod_i`		`(mod)`
`mul_i`		`(*)`
`neg_i`	:: int→int	unary minus
`power_i`	:: int→int→int	integer power
`sub_i`		`(-)`

2.2.10 Double precision floating point functions

There are just two double precision operators, the rest are all functions.

identifier	type	description
`add_d`	:: double→double→double	addition
`arctan_d`	:: double→double	arc tangent
`cos_d`	:: double→double	cosine
`d_to_f`	:: double→float	convert double to float (truncate)
`d_to_i`	:: double→int	convert double to integer (truncate)
`entier_d`	:: double→double	entier
`exp_d`	:: double→double	exp
`log_d`	:: double→double	natural logarithm
`mul_d`	:: double→double→double	multiplication
`neg_d`	:: double→double	unary minus
`power_d`		`(^)`
`sin_d`	:: double→double	sine
`sqrt_d`	:: double→double	square root
`slash_d`		`(/)`
`sub_d`	:: double→double→double	subtraction

2.2.11 Comparison functions

As noted before, the comparison operators are not polymorphic but confined to the type integer. In particular this means that to compare two characters they must first be converted to integers using the `code` function before an integer comparison operator such as `<=` can be applied e.g.

```
is_letter x= code 'A' >= code x & code x <= code 'Z';
```

identifier	type	description
`eq_i`		`(=)`
`ge_i`		`(>=)`
`gt_i`		`(>)`
`le_i`		`(<=)`
`lt_i`		`(<)`
`ne_i`		`(~=)`
`eq_d`	:: double→double→bool	equality
`ge_d`	:: double→double→bool	greater than or equal
`gt_d`	:: double→double→bool	greater than
`le_d`	:: double→double→bool	less than or equal
`lt_d`	:: double→double→bool	less than
`ne_d`	:: double→double→bool	inequality

2.2.12 Bit operations on small integers

The standard bit operations from C are available as functions on integer arguments. Note that names such as `land_x` are reserved.

identifier	type	description
`land_i`	:: int→int→int	bitwise logical and
`lnot_i`	:: int→int	bitwise logical not
`lor_i`	:: int→int→int	bitwise logical or
`lshift_i`	:: int→int→int	left shift first argument by second
`rshift_i`	:: int→int→int	right shift first argument by second

2.2.13 I/O support

Intermediate provides limited I/O support. The formatting functions `show_..` use the C library procedure `sprintf` to format the numeric argument in as little space as possible. Then the string is converted to a list of characters.

The `dynamicread` function takes as argument a list of characters that represents a file name. If the file can be opened successfully for reading, its contents are returned lazily, i.e. character by character as demanded. The end of the file is signaled by the end of the list. If the file cannot be opened the program is terminated with an error message.

The normal mode of operation for a lazy functional program compiled by the FAST compiler is that the main expression (see section 3.3) is evaluated and printed on standard output. This printer driven mode of evaluation is called the "need to print". Sometimes this behaviour is not appropriate, so the function `dynamicwrite` has been provided to enable "output diversions". Like `dynamicread`, it also takes a file name as argument. When the file can be opened successfully for writing, the `dynamicwrite` function returns a one argument function that must be applied to the output which needs to be written out to the file. When this function application is about to become printed by the "need to print", the output will be sent to the opened file instead of standard output. Printing to the previous output stream will be resumed as soon as the end of the output. Output diversions may be arbitrarily nested,

although the operating system will impose a limit on the number of files that may be open at any given time. This mechanism has been copied from the implementation of SASL [15, 16].

identifier	type	description
`show_i`	:: int→[char]	list of characters representing int argument using sprintf format "%d"
`show_f`	:: float→[char]	list of characters representing float argument using sprintf format "%f"
`show_d`	:: double→[char]	list of characters representing double argument using sprintf format "%g"
`show_x`	:: complex_type→[char]	reserved
`dynamicread`	:: [char]→[char]	return a list of characters representing the contents of the file named as argument
`dynamicwrite`	:: [char]→(*→*)	return a unary output diversion function (see text)

2.2.14 Product type construction and access

Intermediate supports tuples, lists and algebraic data types as in most other lazy functional languages. These are implemented by the compiler in terms of just three primitives: `pack` to gather any number of values into a tagged tuple, and `sel` and `tag` to access the components of the tuple. `pack` is strict in its first argument, which will be taken as the tag value of the tuple. The actual argument may thus be an expression, which will be evaluated to an integer before the constructor is created. The different runtime systems available will impose constraints on the actual tag values that may be used, typically to a small positive integer.

Tuples in Miranda and Intermediate are irrefutable, while in most other lazy functional languages tuples are refutable (e.g. Clean, LML and Haskell). Algebraic data types are always refutable (see section 3.12 in the Haskell manual).

It may be instructive to look at some of the intermediate forms a program has to go though when being compiled, to see how these primitives gradually replace the high level pattern matching facilities. (See the `-v` option in section 4 on information how to do this).

These three primitives are available explicitly to allow facilities to be implemented that cannot be properly type checked. The casual user should stay away from these primitives.

identifier	type	description
`pack`	:: int→*1→...*n→(int,*1,...,*n)	Construct an n-tuple with an integer tag given as first argument. The number of remaining arguments determines the arity of the tuple
	(*1 should be read as *, *2 as ** etc.)	
`sel`	:: int→(int,*1,...,*n)→*i	Select a component of an n-tuple. The index must be in the range $1 \ldots n$
`tag`	:: (int,*1,...,*n)→int	Access the tag of an n-tuple

2.2.15 Haskell style array operations

The array facilities of Haskell (see the Haskell manual) are all provided by the FAST compiler. The type of an index must be `int`. In addition a function that is not in Haskell (`tabulate`) was found to be a useful addition. The two constructor functions `assoc` and `descr` are provided to allow the tuples constructed to have strict components. The corresponding access functions are also provided (`indassoc`, `valassoc`, `lowbound` and `upbound`). Rather than using tuple syntax to create or access associations and descriptors one should use these functions, because some code generators implement associations and descriptors not as tuples but as algebraic data types. Arrays are not primitive to Miranda

identifier	type	description
`accum`	:: (*→**→*)→array_type *→ [assoc_type **]→array_type *	See Haskell manual
`accumarray`	:: (*→**→*)→*→descr_type→ [assoc_type **]→array_type *	See Haskell manual (`accumArray`)
`amap`	:: (*→**)→array_type *→ array_type **	See Haskell manual
`array`	:: descr_type→[assoc_type *]→ array_type *	See Haskell manual
`assoc`	:: int→*→assoc_type *	Create an association tuple
`assocs`	:: array_type *→[assoc_type *]	See Haskell manual
`bounds`	:: array_type *→descr_type	See Haskell manual
`descr`	:: int→int→descr_type *	Create an array descriptor
`elems`	:: array_type *→[*]	See Haskell manual
`indassoc`	:: assoc_type *→int	Access the index in an association
`indices`	:: array_type *→[int]	See Haskell manual
`ixmap`	:: descr_type→(int→int)→ array_type *→array_type *	See Haskell manual
`listarray`	:: descr_type→[*]→ array_type *	See Haskell manual
`lowbound`	:: descr_type→int	Access the lower bound of an array descriptor
`subscript`	:: array_type *→int→*	See Haskell manual
`tabulate`	:: (int→*)→descr_type→ array_type *	Create an array consisting of (f l), (f (l+1)), .., (f u)
`upbound`	:: descr_type→int	Access the upper bound of an array descriptor
`update`	:: array_type *→int→*→ array_type *	Similar to the Haskell `(//)` operator but with slightly different arguments
`destr_update`	:: array_type *→int→*→ array_type *	The destructive variant of `update`. Use only if you have done update analysis [4]
`valassoc`	:: assoc_type *→*	Access the value in an association

2.2.16 Operations on complex numbers

The complex constructor function `complex` is strict in its arguments, which greatly improves the performance of programs using such numbers. The built-in complex numbers must not be manipulated by any other than the six functions provided here. Non-strict complex numbers can easily be implemented using floating point arithmetic operations and tuples. Complex numbers are not primitive to Miranda.

identifier	type	description
`add_x`	:: complex_type→ complex_type→ complex_type	Complex addition
`complex`	:: double→double→ complex_type	Construct a complex number
`complex_im`	:: complex_type→double	Select the imaginary component of complex number
`complex_re`	:: complex_type→double	Select the real component of complex number
`mul_x`	:: complex_type→ complex_type→ complex_type	Complex multiplication
`slash_x`	:: complex_type→ complex_type→ complex_type	Complex division
`sub_x`	:: complex_type→ complex_type→ complex_type	Complex subtraction

2.2.17 Parallelism

Three different parallel machines are being built using the FAST compiler, each with its own set of primitives to spark parallel evaluation. The `sandwich` is used on the HyperM system, the `pforce` function is used for a shared memory multi processor and the `pmap` and `procnet` functions are used with the Caliban language. Miranda does not support parallel evaluation.

identifier	type	description
`sandwich`	:: (*1→...*n→*r)→ (*1,int)→...(*n,int)→*r	Evaluate all argument components with types *1 .. *n in parallel, then gather the results using the function supplied as first argument. The integer components are the grain sizes of the jobs [12]
`pforce`	:: (*1,...,*n)→(*1,...,*n)	Evaluate all components of the argument tuple in parallel [20].
`pmap`	:: (*→**)→[*]→[**]	Map a function over a list and evaluate all elements of the result list in parallel
`procnet`	:: [process]→[connection]→ process	Evaluate the processes listed in the first argument on different processors in parallel. The second argument specifies the connections among the processes [10, 5]

To be able to use `procnet` the following type declarations must be made explicitly (i.e. they are not part of advanced Intermediate):

```
process   == [[message]]->[[message]]
connection== ((int,int),(int,int))
message  ::= Char char|Float float ...
```

A connection `((i,j),(m,n))` indicates that output `j` of process `i` should be connected to input `n` of process `m`. Processes are indexed from 1 and process 0 represents the rest of the world (which provides input to the process network and consumes its output). Each process takes as input a list of streams of messages, and delivers a single stream of messages as output.

2.3 Syntax of advanced Intermediate

We use a (highly ambiguous) yacc style grammar to specify the syntax of Intermediate.

2.3.1 Operator precedence

The syntax of advanced Intermediate is close to that of Miranda to ease the conversion from Miranda programs. The yacc grammar that is used by the FAST compiler is therefore heavily based on that of Miranda. The advanced Intermediate grammar is reproduced here with a few slight modifications. Firstly multicharacter literals are enclosed in single quotes just like single character literals. Secondly we use four suffixes to denote:

.opt optional presence,

.star repetition of zero or more times,

.plus repetition of one or more times,

.list one or more items separated by commas, e.g. `a.list` could be "`a`" or "`a,a`" or "`a,a,a`" etc.

A combination of these suffixes is also possible, so `type.list.opt` means a list of `type`, separated by commas or possibly no `type` at all.

The priorities and associativity of the operators are specified below. Priority increases from top to bottom as normal.

yacc key	symbols with the same associativity and priority
%right	`->`
%right	`: ++ --`
%left	`\/`
%left	`&`
%left	`~`
%nonassoc	`> >= = ~= <= <`
%left	`+ -`
%left	`(unary minus)`
%left	`* / mod div`
%right	`^`
%left	`.`
%left	`#`
%left	`!`
%right	`$`

2.3.2 Grammar rules

A program consists of a sequence of function definitions, type definitions and type specifications.

```
program         : /* empty */
                | def  program
                | tdef program
                | spec program ;
```

There are two forms a function definition may take. An ordinary definition specifies a function with at least one argument. The second form of definition is a function without arguments. This is called a CAF (for constant applicative form).

A function may take any number of arguments and the arguments may specify arbitrarily complex patterns. Such patterns may consist of lists, tuples, algebraic data type constructors etc. A function may have more than one defining clause. These are tried in textual order. The arguments of a function are tested from left to right and each argument pattern is tested outside in. For a clause to be chosen all argument patterns present in that clause must match completely. Repeated variables are allowed. The first occurrence is bound to an actual value and subsequent occurrences are matched against the value bound to the first occurrence.

A CAF may specify an arbitrarily complex pattern on the left hand side of its defining equation. The pattern must contain at least one variable but may contain many more. When

one of the variables thus defined is referenced in the program, the entire pattern will be checked for conformity. This applies both to CAFs defined at global level as well as parameterless functions defined local to another function. If the conformity check fails, the program is aborted with an error message. Some languages (e.g. LML) do not require full conformity when only part of the structure covered by the pattern is used.

Function definitions may be preceded by the `INLINE` directive, which will instruct the compiler to substitute the body of the definition for every call, provided the appropriate compiler flags are set (see the `-i` option in section 4).

```
def             : 'INLINE'.opt fnform '=' rhs
                | 'INLINE'.opt pat    '=' rhs ;
```

A type definition may take three forms. The first is a type synonym, which is present for compatibility with Miranda. The second form introduces an algebraic data type. The third form, for an abstract data type, is also present for compatibility with Miranda. The FAST compiler ignores type synonyms and the header of abstract data type definitions, but not the actual definitions in an abstract data type.

```
tdef            : tform '==' type ';'
                | tform '::=' constructs ';'
                | 'abstype' tform.list 'with' spec.plus ';' ;
```

The type of a function may be specified explicitly as an aid in documentation and for compatibility with Miranda. The FAST compiler treats type specifications as comments, so it does not even check the specification other than for syntactic correctness.

```
spec            : tform.list '::' type ';' ;

constructs      : construct
                | construct '|' constructs ;

construct       : constructor argtype.star
                | type '$' constructor type
                | '(' construct ')' argtype.star ;

type            : argtype
                | typename argtype.plus
                | type '->' type
                | type '$' typename type ;

argtype         : typename
                | typevar
                | '(' type.list.opt ')'
                | '[' type.list     ']' ;

tform           : typename typevar.star
                | typevar '$' typename typevar ;

fnform          : var formal.star
                | pat '$' var pat
                | '(' fnform ')' formal.star ;
```

```
pat             : formal
                | '-' natconst
                | constructor formal.plus
                | pat ':' pat
                | pat '+' natconst
                | pat '$' constructor pat
                | '(' pat ')'
                | '(' pat ')' formal.plus ;

formal          : var
                | constructor
                | natconst
                | stringconst
                | charconst
                | '(' pat.list.opt ')'
                | '[' pat.list.opt ']' ;
```

The right hand side of a function definition may contain a number of cases, each accompanied by a guard to allow the appropriate case to be selected. All guarded expressions must be grouped in the last right hand side of a function definition. If a function definition has more than one defining equation, this means that only the last one may have guards. The conditional function `iff` may appear anywhere. At the time of writing, the use of `iff` or `iffrev` will not produce optimal code, so guards should be used whenever possible.

A where expression applies to an entire `rhs` to which it belongs, including the guard expressions in the `rhs`. Use LET expressions to adorn an individual expression with local definitions (Note that Miranda does not allow LET expressions).

```
rhs             : simple.rhs ';'
                | cases ;

simple.rhs      : exp
                | exp whdefs ;

cases           : alt ';' '=' cases
                | lastcase ';' ;

alt             : exp ',' 'if' exp ;

lastcase        : lastalt
                | lastalt whdefs ;

lastalt         : exp ',' 'if' exp
                | exp ',' 'otherwise' ;

whdefs          : 'where' def.plus ;

exp             : e1
                | 'LET' def.plus 'IN' e1
                | prefix1
                | infix ;
```

```
e1              : simple.plus
                | e1 ':'            e1
                | e1 '++'           e1
                | e1 '--'           e1
                | e1 '\/'           e1
                | e1 '&'            e1
                | '~' e1
                | e1 '>'            e1
                | e1 '>='           e1
                | e1 '='            e1
                | e1 '~='           e1
                | e1 '<='           e1
                | e1 '<'            e1
                | e1 '+'            e1
                | e1 '-'            e1
                | '-' e1 %prec UNARY
                | e1 '*'            e1
                | e1 '/'            e1
                | e1 'div'          e1
                | e1 'mod'          e1
                | e1 '^'            e1
                | '#' e1
                | e1 '.'            e1
                | e1 '!'            e1
                | e1 '$' identifier e1
                | e1 '$' IDENTIFIER e1 ;

simple          : var
                | standfun
                | constructor
                | natconst
                | floatconst
                | charconst
                | stringconst
                | '(' infix1 e1 ')'
                | '(' e1 infix ')'
                | '(' pack.exp.list.opt ')'
                | '[' exp.list.opt ']'
                | '[' exp         '..'     ']'
                | '[' exp         '..' exp ']'
                | '[' exp ',' exp '..'     ']'
                | '[' exp ',' exp '..' exp ']'
                | '[' exp '|' qualifs ']' ;

qualifs         : qualifier
                | qualifier ';' qualifs ;

qualifier       : exp | generator ;

generator       : pat.list '<-' exp ;

prefix1         : '~' | '#' ;
```

```
infix           : infix1
                | '-' ;

infix1          : '++' | '--' | ':'
                | '\/' | '&'
                | '>'  | '>=' | '=' | '~='  | '<='  | '<'
                | '+'  | '*'  | '/' | 'div' | 'mod' | '^'
                | '.'
                | '!'
                | '$' identifier | '$' IDENTIFIER ;
```

The terminal symbols of the grammar that are not literally shown are:

terminal symbol	description
`IDENTIFIER`	Identifier beginning with an uppercase letter
`charconst`	Literal character enclosed in single quotes (`'`). For escape sequences see section 2.1
`constructor`	Identifier beginning with an uppercase letter
`floatconst`	Floating point number, thus containing a full stop or an exponent or both
`identifier`	Identifier beginning with a lowercase letter or a standard function name
`natconst`	Natural number (`0`, `1`, ...)
`stringconst`	Literal string enclosed in double quotes (")
`typevar`	Type variable (`*`, `**` ...)
`typename`	Identifier beginning with a lowercase letter
`var`	Identifier beginning with a lowercase letter
`standfun`	Standard function name (see section 2.2)

2.4 Syntax of basic Intermediate

Basic Intermediate is part of the current Haskell- route. It is painful to program in basic Intermediate so we recommend the use of Haskell- or advanced Intermediate instead.

The priorities and associativity of the operators are specified below.

yacc key	symbols with the same priority and associativity
%right	`:`
%left	`|`
%left	`&`
%nonassoc	`> >= = ~= <= < >> <<`
%left	`+ -`
%left	`* / REM DIV`
%right	`**`

Basic Intermediate differs from advanced Intermediate in that it does not support algebraic data types, list comprehensions, arithmetic sequences, syntax for tuples, do-it-yourself infix etc.

```
program         : module ;

module          : defunit
                | defunit module ;

defunit         : 'DEF' defs '?'
                | defs ;

defs            : rule
                | rule  ';' defs ;

rule            : 'INLINE' patlhs '=' letexp
                | patlhs '=' letexp ;

patlhs          : pat
                | ident pats ;

pats            : pat
                | pat pats ;

pat             : 'TRUE' | 'FALSE'
                | char
                | 'NIL'
                | numb
                | ident
                | '(' conses ')' ;

conses          : pat
                | pat ':' conses ;

lets            : letrule
                | letrule ';' lets ;

letrule         : patlhs '=' letexp ;

letexp          : 'LET'    lets 'IN' letexp
                | 'LETREC' lets 'IN' letexp
                | exp ;

exp             : '+' exp %prec '*'
                | '-' exp %prec '-'
                | '~' exp %prec '='
                | exp ':'   exp
                | exp '&'   exp
                | exp '|'   exp
                | exp '+'   exp
                | exp 'DIV' exp
                | exp '='   exp
                | exp '>='  exp
                | exp '>'   exp
                | exp '<='  exp
                | exp '<'   exp
                | exp '>>'  exp
                | exp '<<'  exp
```

```
                | exp '*'   exp
                | exp '~='  exp
                | exp '**'  exp
                | exp 'REM' exp
                | exp '/'   exp
                | exp '-'   exp
                | ifexp
                | apexp ;

ifexp           : 'IF' term term ifexp
                | 'IF' term term term ;

apexp           : term
                | apexp term ;

term            : 'TRUE' | 'FALSE'
                | char
                | 'NIL'
                | numb
                | ident
                | standfun
                | '(' letexp ')' ;
```

The terminal symbols of basic Intermediate that are not literally shown are:

terminal symbol	description
`ident`	Identifier beginning with an any letter
`char`	Literal character preceded by a percent sign (`%`)
`numb`	Any numeric constant
`standfun`	Standard function name (see section 2.2, although some of the functions mentioned there are not available)

3 Restrictions on advanced Intermediate and examples of use

Intermediate is not intended as a programming language but as the target of a frontend. It thus has various shortcomings if regarded as a programming language, which mostly vanish if a proper frontend or program development system is used.

3.1 CAFs and cyclic definitions

CAFs are function applications with constant arguments, where a CAF itself also qualifies as a constant. Special measures are taken to ensure that CAFs are evaluated at most once. This is done as follows. The compiler associates with each CAF a heap cell, to which all uses of the CAF will point. The first time the CAF is evaluated, the associated heap cell is updated with the result. All subsequent references to the CAF will immediately receive the HNF calculated previously.

CAFs may be used to define cyclic data structures but parameterless functions defined local to another function may not be cyclic. There is no restriction of the use of local

function definitions *with* parameters; these may be recursive, mutually recursive etc. and are implemented with the same efficiency as global function definitions. The restriction on local parameterless functions has been applied to make it possible for an efficient reference counting garbage collector to be implemented. This decision also has a disadvantage from the efficiency point of view. Suppose one would like to write a function such as this:

```
main x = xs where xs = x:xs;;
```

The FAST compiler would complain about `xs` not being defined, because its definition is not in scope in the body of `xs`. The compiler could have transformed this program automatically into the semantically equivalent program below, using the standard fixed point function:

```
main' x = xs where xs   = fix f;
                   f xs = x:xs;;
```

The difference between the two versions is, that `main` would develop a cyclic structure, while `main'` would not necessarily do so: that depends on the way `fix` is implemented in the runtime system. A runtime system that uses a two space copying garbage collector such as the performance runtime may well use the cyclic implementation of `fix`, so that the efficiency problem disappears entirely. A runtime system with just a reference counting garbage collector would be well advised not to implement `fix` in a cyclic fashion. Intermediate being a frontend target language we found that to make these considerations visible at the language level would be useful, so that (Intermediate) programmers are aware of the problem and frontend designer can take appropriate action.

3.2 Type inference

The syntax of Intermediate is that of a strongly typed language. In particular the presence of a mechanism to specify algebraic data types and the support for pattern matching on algebraic data types suggest that strong polymorphic type checking will be performed. This however is not the case for the simple reason that in spite appearances, Intermediate still is the target of a frontend compiler, which is supposed to deal entirely with type checking and the generation of meaning full error messages if type checking fails. The FAST compiler allows algebraic data types merely as a convenience and translates them in an early stage of the compilation process into ordinary tuples. The type inferencer that has been implemented in the FAST compiler is not bothered by the type problems generated by this approach. It will thus not be able to generate sensible error messages when real type conflicts occur. The FAST type inferencer is used to infer the types of the basic values that are manipulated, so that the code generators may take advantage of this information. As a consequence it is perfectly possible to use the FAST compiler as a backend for any strongly typed lazy functional language but also for weakly typed languages such as SASL. Also the functions that cannot be implemented in a strong polymorphically type checked language, such as the generic show function, or certain functions necessary to spark parallelism can be implemented efficiently as ordinary functions in Intermediate, using the built in type testing functions where necessary.

3.3 Compilation unit

An Intermediate program must be compiled all at once. This makes it rather difficult to compile large programs (several thousands of lines of source text) on small machines. A separate compilation facility should have been implemented but at present that is not the case. During program development one would nevertheless use the separate compilation facilities of the development system (i.c. Miranda) and when the program is ready to be compiled by FAST, a simple script may combine the relevant modules into a single file.

By convention an Intermediate program contains a function called `main`, which represents the main computation. The three code generator/runtime systems have different requirements for the type of main:

runtime/function type		comments
documentation runtime		
`main`	:: *→[*]	
`input`	:: *	input caf required.
production runtime		
`main`	:: *→[char]	Two possibilities available
`input`	:: *	input CAF required
production runtime		
`main`	:: [char]	Alternative
`input=0;`		dummy input CAF required
performance runtime		
`main`	:: int→...→int→[char]	inputs must all be of type `int`. The actual arguments are taken from the command line, no input CAF(s) required

It should be noted that sensible warnings are not always given when `main` or `input` have incorrect types. Should a runtime error such as "Bus error" or "Segmentation fault" occur, then the first thing to check is whether the types of `main` and `input` satisfy the constraints imposed by the runtime system used.

3.4 Examples of some advanced Intermediate programs

The following example is a complete Intermediate program, which defines two functions and a CAF. Note that each definition is terminated by a semi-colon. The function `ap` is equivalent to the built in `append` function. It is used here merely to illustrate the use of pattern matching.

```
ap :: [*] -> [*] -> [*];
ap []     ys = ys;
ap (x:xs) ys = x : ap xs ys;

main :: [char] -> [char];
main x       = x $ap ".\n";  || This is the same as x ++ ".\n"

input        = "hello world";
```

When compiled and executed the program will print `hello world.` followed by a newline.

The program below is another complete Intermediate program. It defines two show functions and the 6 comparison functions on characters in terms of the standard functions provided for integers. The output of the program is the string "True".

```
show_b :: bool -> [char];
show_b True = "True";
show_b False= "False";

show_c ::char -> [char];
show_c  c   = [c];

c_to_i :: char -> int;
INLINE c_to_i  c = code c;

i_to_c :: int -> char;
INLINE i_to_c  c = decode c;

eq_c, ne_c, gt_c, ge_c, lt_c, le_c :: char -> char -> bool;
eq_c c1 c2  = c_to_i c1  = c_to_i c2;
ne_c c1 c2  = c_to_i c1 ~= c_to_i c2;
gt_c c1 c2  = c_to_i c1 >  c_to_i c2;
ge_c c1 c2  = c_to_i c1 >= c_to_i c2;
lt_c c1 c2  = c_to_i c1 <  c_to_i c2;
le_c c1 c2  = c_to_i c1 <= c_to_i c2;

main x      = show_b (eq_c (i_to_c x) 'A');
input       = 65;
```

The third program is a slightly more interesting example. It demonstrates the use of arrays in the fast Fourier transform [9]. Note that this is not a complete implementation of the FFT because the necessary reorder has been omitted as well as some library functions (see below).

```
complex_array == array_type complex_type;

fft :: int -> int -> complex_array -> complex_array;
fft size 0 xs
       = xs;
fft size n xs
       = fft size (n div 2) xs'
         where
         m  = log2 (size div (n * 2));
         xs'= array (bounds xs)
              (concat [mkpair j | j <- [0..size-1];
                                  j $land_i (1 $lshift_i m) = 0]);
         mkpair j = [assoc j (add_x x z),assoc k (sub_x x z)]
                    where
                    x = subscript xs j;
                    y = subscript xs k;
                    z = mul_x (unitroot size (n*j)) y;
                    k = j + pow2 m;;;
```

```
unitroot :: int -> int -> complex_type;
unitroot i n
       = complex (cos_d phi) (sin_d phi)
         where
         phi = mul_d (i_to_d (2 * n) / (i_to_d i)) pi;;

pow2 :: int -> int;
pow2 x = 1 $lshift_i x;

log2 :: int -> int;
log2 x = d_to_i (entier_d (log_d (i_to_d x) / (log_d 2.0)));

main x = elems (fft 4 2 input);
input  = listarray (descr 0 3) [complex 1.0 2.0, complex 3.0 4.0,
                                complex 5.0 6.0, complex 7.0 8.0];
```

Unlike the first two examples, the `fft` program uses two library functions (`pi` and `concat`) that are not standard functions in Intermediate. Yet these two are not defined in the source. To compile this program the definitions of these two functions will have to be concatenated to the source file. Normally one would concatenate an entire library of standard functions to each source file compiled, and then use the `-r` option (see section 4) to remove the unwanted functions early during the compilation.

4 Using the compiler proper

The FAST compiler proper is embedded in a shell script. There are actually several shell scripts to run the compiler proper, one for each code generator with C compiler and runtime system. The working of these shell scripts is documented in subsequent sections. This section describes the compiler proper, which takes a file with an Intermediate program and generates two files with C-like macro calls: a "header" file with extension `.hdr` and a "code" file with extension `.cde`. The macro calls are normally interpreted by the various code generators to produce a C program that is then compiled by a C compiler and linked with the appropriate runtime system.

The options to the compiler proper do not all have the same default setting as in the shell scripts described later. The shell script defaults override the settings described in this section.

The first set of options govern the program transformations performed by the compiler.

program transformation options		
option format	default	description
`-i n m`	`-i 0 0`	The compiler inlines all functions defined with the `INLINE` directive when $m > 0$. It also inlines those function definitions that are used at most n times. Here a direct recursive call does not count as a use. The second parameter m decides how often the inlining process is to be repeated. By default no inlining takes place. The effects of inlining are best studied by setting option `-v 3`. See also [7]
`-s n`	`-s 0`	The compiler may generate specialised copies of functions that are applied to constant arguments. For example `(map sin_d xs)` will be turned into a new function application `(map_sin_d xs)` where the new function is a copy of the function `map` with `sin_d` substituted for the first argument. The search for constant arguments stops as soon as the first non-constant argument is found, so if the parameters of `map` were reversed, no specialised copy would be generated. By default no specialisation takes place. The effects of specialisation are best studied by setting `-v 3`. See also [7]
`-no_con_fold`	on	The constant folder is capable of applying most standard functions to constant arguments at compile time. User defined functions are not reduced at compile time. One reason is that imported functions other than the standard functions can not be evaluated symbolically. Another is the expense of trying to evaluate user functions such that the compiler does not loop
`-no_caf_lift`	on	CAF lifting causes all constant expressions that are not part of the body of a CAF definition (i.e. a function definition without formal arguments) to become separate CAFs. This is necessary to guarantee laziness

The second set of options applies to the program analysis that is done to gather all the information required to generate good code.

analysis options		
option format	default	description
`-[123]`	`-2`	The domain of the analysis. The compiler performs strictness, cheap eagerness and boxing analysis over a linear domain consisting of a bottom element and 1, 2 or 3 superior elements. At present there is little runtime support for the code generated by any other than the `-1` option
`-no_strict`	on	Strictness analysis can be switched off if desired. This will cause inefficient code to be generated
`-no_eager`	on	Cheap eagerness analysis works out for which functions it is cheaper to call them in a non-strict context than it is to build a suspension for the function. Switching this analysis off does not have a major impact
`-no_box`	on	Boxing analysis decides whether basic data values such as integers and characters need to be stored in the heap or somewhere else (stack, registers). For programs that manipulate large numbers of basic values, boxing analysis has a major impact on the performance.
`-no_box_standard`	on	Many standard functions require unboxed arguments. Selecting this option causes the compiler to assume that all standard functions require boxed arguments. This allows for a simpler runtime system to be built
`-no_private`	on	Privacy analysis works out whether a suspension needs to be updated after it has been evaluated. Selecting this option will require suspensions to be always updated
`-no_typing`	on	Type inference may be switched off if the code generator and runtime system do not require the information. This saves on compilation time
`-sub_typing`	off	For some types it may be useful to have more information than is printed normally. Selecting `-sub_typing` will print types `T` as `T<number>` and similarly types `E` (see also section 7.2)

analysis options		
option format	default	description
`-no_nesting`	on	The macro calls produced on output by the FAST compiler may be arbitrarily nested expressions. This may cause long lines of output. Selecting the `no_nesting` option prints most function calls on a separate line. This will cause many more temporary variables to be declared in the generated code. The performance code generator will give best results if nesting is as deep as possible
`-no_refcount`	on	The compiler is capable of generating calls to a set of functions that increment or decrement the reference count of heap cells. This is intended for use by the reference counting garbage collector. The performance runtime system uses these calls to assist in life time analysis for register allocation. If increment and decrement calls are not used, the `no_refcount` option may be selected to suppress them
`-susp_box_arg`	off	Selecting `susp_box_arg` guarantees that suspensions never contain boxed arguments. This may be convenient for some garbage collectors, but not without cost. The default setting will put unboxed values in suspensions when appropriate, thus saving on boxes (i.e. heap cells)
`-tree`	off	Selecting the `-tree` option will print the flow graph in Intermediate syntax on a separate file, which has the same base name as the Intermediate input file, but extension `tree`

The third set of options tailors the generation of output to the various different code generation and runtime systems available.

output generation control		
option format	default	description
`-mklib`	off	The `mklib` option switches off all knowledge the compiler has about standard functions. This is useful when implementing some of the standard functions in Intermediate. For most standard functions this facility will generate inefficient code
`-r <function>`		This option requires the name of a function (or CAF) in the Intermediate program. It specifies the root function, which is normally `main`. All functions used by the function declared to be the root in this way are compiled, and also the functions they refer to recursively. Functions not reached are discarded. This may reduce compilation time significantly. The `-r` option may be given more than once, so that other functions may be retained as well, e.g. `-r main -r input`. By default no functions are discarded. This option is ignored when the `mklib` option is also specified
`-debug`	off	Compiler debugging. This causes the symbol table to be printed along with some other internal information
`-haskell`	off	Print the syntax tree as a Haskell program as early as possible
`-lml`	off	Print the syntax tree as an LML program as early as possible
`-clean`	off	Print the syntax tree as a Concurrent Clean program after translating list comprehensions and lambda-lifting

Finally here are some miscellaneous options mainly used for debugging purposes:

miscellaneous options		
option format	default	description
`-stop n`	`-stop 0`	Stop after n passes if $n > 0$. This option is used when only a few of the many (over 50) compiler passes are required
`-t n`	`-t 0`	Used for debugging purposes
`-v n`	`-v 0`	Verboseness, $v = 0$ means no redundant output, $v = 1$ gives a short report on some properties of the compiled program, $v = 2$ also lists all the compiler passes when they are run, $v = 3$ prints the entire syntax tree just before it is transformed into a flow graph, $v = 4$ and $v = 5$ prints the syntax tree after several other passes as well. Selecting `-v 3` or higher also lists all properties of the standard functions such as the boxity strictness, privacy and type of argument positions and the cheap eagerness of the function result on standard output. Increasing values give more and more output

There are two positional parameters, which may appear anywhere in the list of options. The first non-option is taken to be the input file and the second the dump file:

positional parameters		
file name	status	description
`<file>.i`	mandatory	File with program to be compiled (recommended extension `.i`)
`<file>`	optional	File to receive the flow graph dump pertaining to the program being compiled. This file is intended for debugging purposes only

The output produced by the compiler has the same file name as the input file. The `.i` suffix is replaced by `.cde` for the "code" file, which contains the macro calls that will eventually lead to a C program. The second file, the "header" file, contains many useful declarations that may be turned into a C header file. It has a suffix `.hdr`.

4.1 Passes of the compiler proper

Specifying values greater than one as an argument to the `-v` option will print the compiler passes as they are run. The following is a list of the most important passes, with a short explanation of their purpose. The second column gives the compiler option (if any) that may be used to control the execution of the pass. The passes are listed in the order of execution, but some passes may be run more than once.

passes operating on the syntax tree		
pass	option	description
read	always	Read, parse and slightly simplify the input. This includes translation of infix operators to prefix function applications and translation of where expressions into LET expressions
check	always	Check for undefined identifiers
haskel	`-haskell`	Pretty print the parse tree in Haskell syntax
lml	`-lml`	Pretty print the parse tree in LML syntax
compr	always	Translate list comprehensions
lift	always	Perform lambda lifting
clean	`-clean`	Pretty print the parse tree in Concurrent Clean syntax
tdef	always	Generate constructor functions from the algebraic data type declarations
cafpat	always	Translate conformal patterns (in LET definitions and CAFs) into conditional expressions
funpat	always	Translate pattern matching on function arguments into conditionals expressions
letdan	always	Delete unused LET defined identifiers and inline those that are used only once
lethie	always	Float LET definitions as far into the body of the surrounding function as possible
spec	`-s`	Specialise functions with constant arguments
inline	`-i`	Inline functions adorned with INLINE directive and based on a heuristic
prune	`-r`	Prune unused functions

The first passes of the compiler operate on a classical syntax tree representation. The program analysis phases operate on a representation that we call flow graphs [6].

flow graph preparation for the analyses		
pass	option	description
flow	always	Transform the abstract syntax tree into a flow graph
gat	always	Gather dangling edges and insert SINK nodes
con	`-no_con_fold`	Constant folder
kil	`-no_con_fold`	Kill the redundant nodes that may have been introduced by the constant folder
use	always	Count the number of USE nodes emanating from a single edge
fun	always	Discover the edges in the flow graph that carry functions
abs	always	Abstraction propagation and generation of CALL nodes
lit	always	Discover which edges carry literals
caf	`-no_caf_lift`	Perform CAF lifting
jmp	always	Discover tail talls
typ	`-no_typing`	Gather type information
uni	`-no_typing`	Unify type information
zip	always	Work out where lists are split in head and tail field so that they may be "zipped" up again

The proper analysis phases are each using four passes. After collecting some information about the structure of the program (1), a set of equations is built (2), that must be solved (3) so that the results can be substituted back into the flow graph as a set of attributes (4) [6].

flow graph analyses		
pass	option	description
std	`-no_strict`	Gather dependency information for strictness
prestr	`-no_strict`	Prepare dependency equations for strictness
solve	`-no_strict`	Solve dependency equations for strictness
str	`-no_strict`	Insert strictness information into the flow graph
egd	`-no_eager`	Gather dependency information for cheap eagerness
preegr	`-no_eager`	Prepare dependency equations for cheap eagerness
solve	`-no_eager`	Solve dependency equations for cheap eagerness
egr	`-no_eager`	Insert cheap eagerness information into the flow graph
nfm	always	Combine cheap eagerness and strictness into normal form information
ini	always	Prepare initialisation of static heap cells
bxd	`-no_box`	Gather dependency information for boxing
prebox	`-no_box`	Prepare dependency equations for boxing
solve	`-no_box`	Solve dependency equations for boxing
box	`-no_box`	Insert boxing information into the flow graph
prd	`-no_private`	Gather dependency information for privacy
preprv	`-no_private`	Prepare dependency equations for privacy
solve	`-no_private`	Solve dependency equations for privacy
prv	`-no_private`	Insert privacy information into the flow graph

The output generation is now in principle pretty printing. There is quite a lot of bookkeeping involved so a considerable number of passes are required still.

output generation		
pass	option	description
ref	always	Compile time reference count
val	always	Generate expressions and values
tmp	always	Allocate temporary variables
ynk	always	Discard unused versions of functions
hdr	always	Print the header file with declarations
tree	`-tree`	Print the flow graph in Intermediate syntax on a separate file, which has the same base name as the Intermediate input file, but extension `tree`
import	`-v 3`	List properties of standard functions, such as strictness, boxity, privacy and type on standard output. This also works with `-v 4` etc.
ord	always	Order the nodes in the flowgraph
mod	always	Print the code file
inc	`-no_refcount`	Print increment instructions to the code file
dec	`-no_refcount`	Print decrement instructions to the code file
check	`-debug`	Check the flow graph for consistency

4.2 Compiler output

The compiler output has the form of a series of macro calls. A macro may have an arbitrary number of parameters, separated by commas and enclosed in parentheses. Using a reasonably sophisticated macro processor (`m4` rather than `cpp`) it should be possible not only to generate C, but also Pascal, Modula-2 etc from the macro calls.

For a discussion of the various forms of identifier names see [7] and section 7. The macro calls that may be found in the "header" file are:

```
hdr  : 'hdr_begin'    '(' id   ')'
     | 'hdr_version'  '(' ...  ')'
     | 'hdr_end'      '(' id   ')'
     | 'hdr_prel'     '(' id   ')'
     | 'hdr_type'     '(' type ')'
     | 'hdr_imp_fun'  '(' id   ')'
     | 'hdr_imp_prel' '(' id   ')'
     | 'hdr_imp_caf'  '(' id   ',' id     ')'
     | 'hdr_imp_proc' '(' id   ',' id     ')'
     | 'hdr_caf'      '(' id   ',' id     ')'
     | 'hdr_numb'     '(' id   ',' float  ')'
     | 'hdr_numb'     '(' id   ',' nat    ')'
     | 'hdr_char'     '(' id   ',' char   ')'
     | 'hdr_string'   '(' id   ',' string ')'
     | 'hdr_cons'     '(' id   ',' cons   ')'
     | 'hdr_proc'     '(' nat  ',' id     ',' type     ids ')'
     | 'hdr_fun'      '(' id   ',' id     ',' id   ',' nat ')' ;
```

```
ids  : /*empty*/
     | ',' id.list ;

cons : '_11_CONS'     '(' id   ',' id      ')' ')'
     | '_01_CONS'     '(' id   ',' id      ')' ')'
     | 'bind'         '(' id   ',' id      ')' ')'
     | 'vap'          '(' nat  ',' id.list ')' ')' ;
```

The terminal symbols of the grammar that are not literally shown are:

terminal symbol	description
`id`	identifier
`char`	literal character enclosed in single quotes (')
`float`	optionally signed floating point number with a suffix `_D`
`int`	optionally signed integer number with a suffix `_I`
`nat`	unsigned natural number without any suffix
`string`	literal string enclosed in double quotes (")
`type`	type string (see section 7.2)

The `hdr_proc` macro provides all the information that the user may care to inspect to see how successful the analysis of the compiled program has been. The first parameter indicates whether the function was found to be eager (if the parameter value is 1, 0 otherwise). A function is eager if it produces a cheap weak head normal form when all strict arguments are supplied as weak head normal forms. An example is `add_i`. Counter examples are `iff`, `hd` and `tl`. The second parameter of the macro is the name of the compiled function, the third is its type and the remaining parameters are names of its arguments. Each argument is adorned with a three digit specification of the form `_br_name_p`. If the `r` digit is 1, the argument is head strict. If the `b` digit is 1, the argument is required in unboxed form, in boxed form otherwise. Only strict arguments can be required in unboxed form, so the combination `_10_` can never happen. If the third (trailing) digit `p` is 1, the argument is private. This means that if ever there is a need to build a suspension for the current argument position, the suspension does not require an update after it has been evaluated. If the privacy of an argument is 0, an update is required or laziness may be affected.

The header file `hw.hdr` as generated for the hello world program is shown below.

```
hdr(_99_,9, hdr_begin(_hw))
hdr(_99_,8, hdr_version(28,1,1,0,1,susp_box_ret,susp_unb_arg,
                        no_con_fold,normal_typing,no_nesting))
hdr(_99_,8, hdr_type(LT))
hdr(_99_,8, hdr_type(LC))
hdr(_99_,8, hdr_type(B))
hdr(_99_,5, hdr_proc(1,_01_input,LC))
hdr(_99_,5, hdr_proc(0,_11_main,LC,_11_x_LC_0))
hdr(_99_,5, hdr_proc(0,_11_ap,LT,_11_1A_LT_0,_00_ys_LT_1))
hdr(_99_,5, hdr_proc(0,_01_ap,LT,_11_1A_LT_0,_00_ys_LT_1))
hdr(_99_,4, hdr_string(_11_257_LC_,"hello world"))
hdr(_99_,4, hdr_string(_11_256_LC_,".\n"))
hdr(_99_,4, hdr_string(_01_257_LC_,"hello world"))
hdr(_99_,4, hdr_string(_01_256_LC_,".\n"))
```

```
hdr(_99_,3, hdr_caf(_01_input,_input))
hdr(_99_,2, hdr_prel(_1111LT_0))
hdr(_99_,2, hdr_prel(_1100LT_0000LT_0))
hdr(_99_,1, hdr_fun(_01_ap_1100LT_0000LT_0_2,
                    _01_ap,_1100LT_0000LT_0,2))
hdr(_99_,1, hdr_fun(_01_TL_1111LT_0_1,_01_TL,_1111LT_0,1))
hdr(_99_,1, hdr_fun(_01_HD_1111LT_0_1,_01_HD,_1111LT_0,1))
hdr(_00_,0, hdr_end(_hw))
```

The macro calls that may appear in the "code" files are:

```
cde  : 'mod_begin'   '(' id       ')'
     | 'mod_end'     '(' id       ')'
     | 'mod_version' '(' ...      ')'
     | 'begin'       '(' id       ')'
     | 'end          '(' id       ')'
     | 'if'          '(' exp      ')'
     | 'else'        '(' exp      ')'
     | 'fi'          '(' exp      ')'
     | 'init'        '(' id.list  ')'
     | 'local'       '(' id.list  ')'
     | 'assign'      '(' id       ',' exp  ')'
     | 'proc'        '(' id       ',' type     ids ')'
     | 'args'        '(' id       ',' type     ids ')'
     | 'trace'       '(' id       ',' type     ids ')'
     | 'dec'         '(' id       ',' type ',' nat ')'
     | 'inc'         '(' id       ',' type ',' nat ')'
     | 'return       '(' id       ',' type ',' exp ')' ;

exp  : id
     | id            '(' exp.list ')' ;
```

An excerpt of the code file `hw.cde` generated for the hello world program is:

```
mod(_99_,9,        mod_begin(_hw))
mod(_99_,8,        mod_version(28,1,1,0,1,susp_box_ret,susp_unb_arg,
                               no_con_fold,normal_typing,no_nesting))
mod(_11_main,96,   proc(_11_main,LC,_11_x_LC_0))
mod(_11_main,95,   args(_11_main,LC,_11_x_LC_0))
mod(_11_main,93,   begin(_11_main))
mod(_11_main,91,   trace(_11_main,LC,_11_x_LC_0))
mod(_11_main,1,    return(_11_main,LC,
                          _11_ap(_11_x_LC_0,_01_256_LC_)))
mod(_11_main,0,    end(_11_main))
mod(_11_ap,96,     proc(_11_ap,LT,_11_1A_LT_0,_00_ys_LT_1))
mod(_11_ap,95,     args(_11_ap,LT,_11_1A_LT_0,_00_ys_LT_1))
mod(_11_ap,94,     local(_0X_248_LT,_0X_238_LT,_0X_228_T,_1X_214_B))
mod(_11_ap,93,     begin(_11_ap))
mod(_11_ap,92,     init(_0X_248_LT,_0X_238_LT,_0X_228_T,_1X_214_B))
mod(_11_ap,91,     trace(_11_ap,LT,_11_1A_LT_0,_00_ys_LT_1))
mod(_11_ap,82,     assign(_1X_214_B,_11_NULL(_11_1A_LT_0)))
mod(_11_ap,7,      if(_1X_214_B))
mod(_11_ap,6,      return(_11_ap,LT,_11_box_red_00(_00_ys_LT_1)))
```

```
mod(_11_ap,5,      fi(_1X_214_B))
mod(_11_ap,44432,  assign(_0X_228_T,
                          vap(2,_01_HD_1111LT_0_1,_11_1A_LT_0)))
mod(_11_ap,4434432,assign(_0X_238_LT,
                          vap(2,_01_TL_1111LT_0_1,_11_1A_LT_0)))
mod(_11_ap,4432,   assign(_0X_248_LT,vap(3,_01_ap_1100LT_0000LT_0_2,
                                         _0X_238_LT,_00_ys_LT_1)))
mod(_11_ap,1,      return(_11_ap,LT,_11_CONS(_0X_228_T,_0X_248_LT)))
mod(_11_ap,0,      end(_11_ap))
mod(_01_input,96,  proc(_01_input,LC))
mod(_01_input,95,  args(_01_input,LC))
mod(_01_input,93,  begin(_01_input))
mod(_01_input,91,  trace(_01_input,LC))
mod(_01_input,1,   return(_01_input,LC,_01_257_LC_))
mod(_01_input,0,   end(_01_input))
...
mod(_00_,0,        mod_end(_hw))
```

More information about the format and intended purpose of the macro calls may be found in [7] and chapter 7.

4.3 Installing the FAST compiler and the documentation runtime

The advanced Intermediate FAST compiler and the documentation runtime system are distributed as a compressed tar file, which should be uncompressed and unpacked in an empty directory. The directory structure created by `tar` in that directory is shown in in the left column below. The source files are listed in the middle column and the files compiled from the sources are shown to the right. The production runtime system is supplied separately.

directory	source files	compiled files
./fast	make.sh, lint.sh	
./fast/input	Makefile, *.c, *.h, *_dir,	*.o,
	lexinterm.l,	lexinterm.c,
	yaccinterm.y	yaccinterm.c
./fast/let	Makefile, *.c, *.h, *_dir	*.o
./fast/pat	Makefile, *.c, *.h, *_dir	*.o
./fast/spec	Makefile, *.c, *.h, *_dir	*.o
./fast/flat	Makefile, *.c, *.h, *_dir	*.o, fast
./fast/nonflat	Makefile, *.c, *.h, *_dir	*.o, fast
./fast/refcount	Makefile, *.c, *.h, *_dir	*.o, fast
./fast/runtime	Makefile, *.c, *.h, *_dir	*.a
./fast/runtime/bin	*cb	
./lib	make.sh, source,	fast2*.*, Fast2.*,
	stdenv.lit.m, Word.hs	TEST.*, ghclib/*, hbclib/*
./bin	fast, m2i	

To compile the compiler and the documentation runtime system execute the commands below.

```
% cd fast
% sh -x make.sh >& make.log
% cd runtime
% sh -x make.sh >& make.log
```

Please note: if the version of `yacc` you are using generates a parser that allocates its stack statically then you will want to keep the define `-DFREE_YACC_STACK` in the file `Makefile` for the compilation of `read_input.c` in directory `fast/input`. If the generated `yacc` parser uses `malloc` and `free` to allocate and deallocate the stack, then you must remove the define.

Executing the commands above will create three binaries, all named `fast`, but in different directories:

fast/refcount/fast This binary contains all available code. It is used by the `fast` shell script described in section 5 to compile advanced Intermediate programs.

fast/nonflat/fast This binary is used for debugging purposes only. The difference with the binary in the `refcount` directory is that the binary in the `nonflat` directory does not generate increment and decrement instructions to support a reference counting garbage collector or life time analysis of local variables.

fast/flat/fast This binary is used to compile basic Intermediate programs.

The two scripts in the `./bin` directory require editing because they need to know in which directory the compiler binary is, and where the runtime library is. The file `make.sh` in the directory `./lib` needs to be modified as well so that it is able to find the two scripts. After that has been done, the library can be compiled using the following two commands:

```
% cd lib
% sh -x make.sh >& make.log
```

This script will generate several error messages unless it has access to the Chalmers Haskell compiler `hbc`, the Glasgow Haskell compiler `ghc`, the Chalmers LML compiler `lmlc`, the Nijmegen Concurrent Clean compiler `clm`, the Miranda system `mira` and the performance code generator (`fcg`, `macro` etc.). The FAST compiler can be used quite happily without compiling the library. In that case Intermediate programmers have access only to the built in functions.

5 Using the documentation runtime system

The FAST compiler proper for advanced Intermediate is embedded in a shell script that runs the appropriate UNIX commands to compile an Intermediate program to an executable file. The shell script (called `fast`) accepts most of the compiler proper options described in section 4, although some have a different default setting. This depends on the code generator selected. In addition the `fast` script passes all options that it does not recognise to the code generator selected. To find out exactly what options are presented to which command, select the `-x` option.

The C code generators available with advanced Intermediate are the documentation and performance generators. The production generator has its own shell script (see section 6). It is also possible to compile advanced Intermediate using several other compilers for lazy functional languages as code generator (LML, Haskell and Concurrent Clean).

option format	default	description
`-DEBUG`	off	Generate code to produce a trace of most function calls at runtime
`-KEEP`	off	Keep all temporary files. This is useful if the code generator, C compiler or linker produce an error message
`-STAT`	off	Arrange for the executable to print some runtime statistics
`-c`	off	Do not call the code generator and C compiler
`-clean`	off	Generate a Concurrent Clean program and call the Concurrent Clean compiler to produce an executable. This requires software from others
`-fcg`	off	Use the performance code generator and runtime to produce an executable. This requires software from others
`-fast`	on	Use the documentation code generator and runtime to produce an executable
`-i n m`	`-i 0 1`	See section 4
`-ghc`	off	Generate a Haskell program and call the Glasgow Haskell compiler to generate an executable. This requires software from others
`-hbc`	off	Generate a Haskell program and call the Chalmers Haskell compiler to generate an executable. This requires software from others
`-lml`	off	Generate an LML program and call the Chalmers LML compiler to generate an executable. This requires software from others
`-o executable`	see text	This option specifies the name of the executable. By default this is made up of the base name of the input file name and one of `.fast.out`, `.fcg.out`, `.clean.out`, `.ghc.out`, `.hbc.out` or `.lml.out` as suffix, depending on the code generator used
`-x`	off	print each command executed during the compilation process. This is useful to find out what is going on when things go wrong

5.1 Runtime options to the documentation runtime system

At runtime there are also some options available. Some of these will only have effect if the appropriate compile time options have been selected as well. The most important option is the `-r` option, which tells the runtime system what kind of argument the main function expects (see [7]). The compiler normally prints this information as a line beginning with `mod(`. The information can have one of the four forms shown in the first column of the table below. The corresponding option that must be specified is shown in the second column:

macro call output by the compiler	option
`mod(_11_main,..., proc(_11_main,...,_11_...))`	`-r 1111`
`mod(_11_main,..., proc(_11_main,...,_00_...))`	`-r 1101`
`mod(_11_main,..., proc(_11_main,...,_01_...))`	`-r 1101`
`mod(_11_main,..., proc(_11_main,...,_0X_...))`	`-r 1101`

This mechanism is rather painful, because if the wrong `-r` option is selected, a Bus error or Segmentation fault will result. The mechanism should have been compiled into the code, but at present that is not the case. The production runtime system uses exactly the same option and correspondence.

Here are the options that can be given to the documentation runtime system:

option format	default	description
`-h n`	`-h 100000`	Set the size of the heap to n (32-bit) words. At present the documentation runtime system has no garbage collector, so selecting a small heap will cause most programs to crash. The two other runtime systems do have a garbage collector, so they will also accommodate large programs
`-r nnnn`	`-r 1111`	The `-r` option requires as argument 4 digits in the range 0..1. The first two are always 1. The next two vary depending on the first two digits in the argument to `_11_main` as shown in the table above
`-l`	off	Print all constructors to reveal the structure of the output list
`-s`	off	Print function call and cell claim statistics upon termination. This requires the `-STAT` or `-DEBUG` compilation option
`-ss`	off	Print function call and cell claim statistics upon termination and also print the elapsed time after each element of the output list has been printed. Requires `-STAT` or `-DEBUG`
`-sss`	off	Print function call and cell claim statistics after each element of the output list has been printed. Requires `-STAT` or `-DEBUG`
`-R n`	`-R 1`	Execute the main program n times. This is useful for performing time measurements

5.2 Example- fully optimised execution

Consider the hello world program from the section examples and suppose it is stored in a file `hw.i`. The following two commands compile and execute the program with the documentation runtime system (`-r 1111` is the default):

```
% fast hw.i
mod(_11_main,96, proc(_11_main,LC,_11_x_LC_0))
```

```
% hw.fast.out
_11_main _11_input:
hello world.
0.07    user + system seconds for hw.fast.out
```

The only output line from the compilation prints the procedure header macro for the main program. The name of the C function that corresponds to the main program is `_11_main`. This functions is of type `LC` which is short for List of Character. Its argument `_11_x_LC_0` is also a list of characters. For a further explanation of the significance of these digits refer to [7] and to section 7.

When the program is executed, the runtime system prints the name of the main function and the name of its argument, then the real output, which is followed by a printout of the cpu time consumed. The default setting of the options to the `fast` script generate fully optimised code, without any form of runtime checking. In particular this means that heap overflow is not detected by the runtime system but instead draws an unhelpful message from the operating system (e.g. Bus error or Segmentation fault).

5.3 Example - execution with statistics and runtime checking

Compiling the same program again with statistics and also runtime checking enabled yields output, which looks like the following when all redundant information is taken out:

```
% fast -STAT hw.i
mod(_11_main,96, proc(_11_main,LC,_11_x_LC_0))
% hw.fast.out -s
_11_main _11_input:
hello world.
box   unbox total user and prelude function call statistics
11    1     12    ap
1     0     1     input
0     1     1     main
11    0     11    prel_1100LT_0000LT_0
23    2     25    total user and prelude functions
box   unbox total list function call statistics
10    1     11    CONS
0     12    12    NULL
10    13    23    total list functions
..
box   unbox total cell claim statistics
10    11    21    CONS
11    0     11    VAP
21    11    32    total (130 ints heap space used)
0.1   user + system seconds
```

The first column above shows the number of times the boxed version of a function is called. The second column shows the number of times the unboxed version is called and the third column shows the total. For instance the function `ap` is called 12 times, which can be accounted for as follows: the number of characters in the string `hello world` is 11, and one extra call is necessary to deal with the trailing `[]`. The bottom lines marked `CONS` and `VAP` relate to cell claims rather than function calls.

Note that to compile the code that generates statistics the `-STAT` compiler option is required, and to actually print the statistics, the runtime option `-s` must also be given. To enable runtime checking, compilation with `-STAT` is sufficient. In that case heap overflow and other similar errors will draw a meaning full message.

5.4 Example - execution with a runtime trace

Compiling the same program with debugging enabled yields rather a lot of output because most functions that are called produce one line, with some details about their actual arguments. The trace consist of two sections. The first is printed in response to all the static declarations. For example the CAF `input` is allocated to a heap cell at address `26548`. The second section is the actual trace, which begins with a call to the main function `_11_main`. The trace is printed up to the point where the first character `h` of the regular output appears at the beginning of the last line. Prominent in the trace are the calls to `_reduce`, which evaluates expressions that have been stored in the heap for later use to HNF.

```
% fast -DEBUG hw.i
mod(_11_main,96, proc(_11_main,LC,_11_x_LC_0))
% hw.fast.out
heap  100000 ints from 30948 to 923c4
                 version: 28,1,1,0,1,susp_box_ret,susp_unb_arg,
                          no_con_fold,normal_typing,no_nesting
             _11_257_LC_: string=hello world at 26528
             _11_256_LC_: string=. at 26530
                  _input: caf=2924 busy=FALSE at 26548
...
      _11_main _11_input:
                  reduce: caf=26548
               _01_input:
                  update: root=26548 top=26538 tag=4
                _11_main: _11_x_LC_0=26528
                  _11_ap: _11_1A_LT_0=26528 _00_ys_LT_1=26540
                    vap3: fun=26554,a1=309e0,a2=26540 at 309f8
                _11_CONS: h=26a34 t=309f8 at 30a08
                  reduce: box=26a34 tag=3 data=68
h                 reduce: vap=309f8
```

The `-DEBUG` option should only be used as a last resort, because when something goes wrong one should go back to the program development system and look for type conflicts, programming errors etc. The `-DEBUG` option is most useful for debugging the runtime system and the compiler.

5.5 Example - execution with a stack trace

Since the FAST compiler generates an ordinary C program, it is also possible, and often quite useful to compile the C sources with the `-g` option. This option is automatically selected when either `-STAT` or `-DEBUG` are selected. The appropriate library is also compiled with the `-g` option. It is thus possible to obtain a stack traceback of the program when it fails using a standard debugger such as `dbx`. The compiler makes an effort to use as many of the names

originally appearing in the Intermediate program, so that the information provided by `dbx` is generally quite useful.

To demonstrate the procedure the hello world program has been modified slightly to make it go wrong. The first defining equation for `ap` has been commented out, so that there is no case to deal with an empty list:

```
ap :: [*] -> [*] -> [*];
|| ap []     ys = ys;
ap (x:xs) ys = x : ap xs ys;

main :: [char] -> [char];
main x       = x $ap ".\n";  || This is the same as x ++ ".\n"

input        = "hello world";
```

Here is the transcript of compiling and executing the program, using `dbx` to discover what went wrong. All runtime error messages will call the procedure `runtime_exit`, so a breakpoint must be set at this point to enable a stack trace to be printed:

```
% fast -STAT hw.i
mod(_11_main,96, proc(_11_main,LC,_11_x_LC_0))
% dbx hw.fast.out
Reading symbolic information...
Read 5037 symbols
(dbx) stop in runtime_exit
(2) stop in runtime_exit
(dbx) run -r 1111
Running: hw.fast.out -r 1111
stopped in runtime_exit at line 70 in file "runtime.c"
   70   (void) exit(code);
(dbx) where
runtime_exit(code = 3), line 70 in "runtime.c"
_01_ABORT(b = 0x221f8), line 111 in "standard.c"
_01_LIT259(), line 83 in "hw.mod.c"
reduce(root = 0x221ec), line 173 in "reduce.c"
_01_ap(_11_1A_LT_0 = 0xffffffff,
       _00_ys_T_1 = 0x221d8), line 72 in "hw.mod.c"
prel_1100LT_0000T_0(proc = &_01_ap() at 0x24f4,
                    arg = 0x2ab58), line 16 in "hw.prel.c"
reduce(root = 0x2ab50), line 161 in "reduce.c"
main(argc = 3, argv = 0xf7fff954), line 266 in "runtime.c"
```

In the stack traceback the function `_01_ap` appears with as its first argument `0xffffffff`, which is the constant used for `[]`. Please refer to the sources in the directory `fast/runtime` for more information about representation and other issues.

5.6 Example - execution with profiling

The C sources generated by the FAST compiler can also be profiled using the standard UNIX profiling tools. This requires compilation of the C sources with the `-p` option. At present none of the runtime libraries are compiled with the `-p` option, so the profiling information

about the library functions is at the same level as the ordinary C runtime library functions. As with the `dbx` the use of original names makes profiling a practical tool.

To demonstrate profiling, the hello world program has been modified to make it use more time. The original program spends 100% of its time in the system function `write`, which is not interesting. The definition of the input list has been changed to make the list 10000 characters long:

```
ap :: [*] -> [*] -> [*];
ap []     ys = ys;
ap (x:xs) ys = x : ap xs ys;

main :: [char] -> [char];
input        = ['X' | i <- [1..10000]];
```

Here is an excerpt of output obtained from compiling and running the modified program:

```
% fast -p hw.i
mod(_11_main,96, proc(_11_main,LC,_11_x_LC_0))
% hw.fast.out -h 1000000
% prof hw.fast.out /home/pieter/profiles/9576.hw.fast.out
 %time  cumsecs  #call  ms/call  name
  40.0     0.04                  _reduce
  20.0     0.06  10004     0.00  __doprnt
  10.0     0.07  10001     0.00  __01_1F_1C_input
  10.0     0.08                  __01_FROMTO_I
  10.0     0.09  10000     0.00  _prel_1100LT_0000LT_0
  10.0     0.10                  mcount
...
```

As is common for lazy functional programs, most time is spent in `_reduce`. Because the runtime system has not been compiled with `-p`, the number of times the runtime support functions are called does not appear in the statistics. `_doprnt` is the C-library procedure responsible for formatting output. The function `__01_1F_1C_input` is generated from the `input` CAF to implement the list comprehension. `_01_FROMTO_I` is the runtime equivalent of the arithmetic sequence in the program. The example suggests that the main program does not appear to use up any time. This is misleading, because some of the work in `main` involved is actually attributed to `reduce`.

6 Using the production runtime system

The production code generator and runtime system can be used to compile lazy functional programs into runnable executables, via instrumented C code. The `fastrts` directory contains the source code for the production runtime system library and the associated C code generation phase. The preferred way to generate an executable is via a front end of some sort, typically Haskell-. It is also possible to build an executable from an Intermediate (.i) file using make.

A `makefile` is provided, and this contains rules for building a binary from an arbitrary Intermediate program file. The system may also be driven through a shell script, `f92`, which accepts a program written in Intermediate form, and produces a C program file. To produce an executable, this must then be linked with the production runtime system either by hand, or with the `makefile`.

6.1 Installation

Assuming you are in the `fastrts` directory, using `csh` then:

```
% limit stacksize unlimited
% setenv fastdir `pwd`
% make
```

may be used to build the runtime system. The `limit stacksize` command prevents C stack overflow when runtime checks are enabled on highly recursive programs.

The environment variable `fastdir` needs to be set to an appropriate `fastrts` directory before compiling programs. This allows a different version of the system to be executed by changing this one environment variable.

6.2 A basic Intermediate example program

Each program file should contain a CAF called `main`, which evaluates to a list of characters. Executing the program causes the expression `main` to be evaluated and printed. The following basic Intermediate example program implements a lazy stream based filter which removes all occurrences of the character "e" from its input stream:

```
|| Program drop_e.i:
|| Remove all occurrences of character ``e'' from standard input.
DEF
input=NIL; || input must be defined, even if never used...

filter f NIL = NIL;
filter f (a:b) = IF (f a) (filter f b) (a:filter f b);

is_e %e = TRUE;
is_e  _ = FALSE;

main = filter is_e (dynamicread NIL)?
```

To execute this example, ensure that a copy of the default `makefile` is in the current directory:

```
% cp $fastdir/makefile .
```

Now type `make`, followed by the name of your Intermediate file, without the extension. An implicit rule in the makefile tells make how to build binaries from Intermediate files:

```
% make drop_e
/home/john/fastrts/f92 -1 drop_e.i
module drop_e.t, 7 functions, 8 arguments
total 4 functions, 3 arguments (inliner run 1 times)
total 4 functions, 3 arguments (prune run 3 times)
    2 head strict  arguments
    2 head eager   functions
    2 head unboxed arguments
    1 head private arguments
  136 nodes 1.50 cpu.sec. 29 maj.flts. 299 pages max.rss
```

```
Translating to C......
gcc -traditional -g -I/home/john/fastrts -DGC_STATS -DFLUSH
    -DALLCHECKS  -DSTATS -DSLOWCONV -DMALLOCGC
    -D_ARRAY_TAIL=_11_TAIL -sun4 -c drop_e.c
```

Executing the compiled program causes `main` to be evaluated and printed:

```
% drop_e
This is me typing to test the drop_e program.
This is m typing to tst th drop_ program.
^D
%
```

6.3 An alternative execution method

A more complicated, potentially more efficient method of invoking programs is possible. Instead of defining `main` as a CAF, it may be defined as a one argument function, with `input` defined as the parameter to `main`. In this case the compiler generates C code which applies main to input, printing the result. For example:

```
% cat foo.i
DEF
main x= 2 + x;
input= 3?
% make foo
/usr/fastrts/f92 -1 foo.i
module foo, 2 functions, 1 arguments
1 head strict  arguments
2 head eager   functions
1 head unboxed arguments
1 head private arguments
Generating C....
gcc -I/usr/fastrts -DAST -DGC_STATS -DPOPMACROS -DBOXITY_CHECK
    -DFLUSH -DTCK -DTCHK -DCHK -DPRIV_CHECK -DLAZLOG -DSTATS
    -DTRACEABLE -sun4 -c foo.c
```

Depending on the outcome of the analysis, the binary may need an argument of `-r 1111` or `-r 1101`. If the function main was found to be strict in it's argument, then the `-r 1111` option is needed to supply the input to main in unboxed, evaluated form:

```
% foo -r 1111
5
```

The outcome of the analysis may be found by using grep to search for the compiled main function in the generated C output.

6.4 Experimenting with the system

The system allows the user to experiment with various analyses and runtime parameters at various stages up to the point of execution:

- C code generation,
- C code compilation,
- runtime options.

6.4.1 C code generation

In the past, the most frequently changed parameter to the compiler proper was the level of strictness and boxing analysis performed (typically flat or non flat) and so this is now setup in the first non comment lines of the `makefile` used above. Other parameters may be changed by editing the shell script `f92`, which contains a line defining:

```
fastflags=''-v 1 $analysis -r main -r input -s 0 -no_nesting''
```

This string is passed to the compiler proper, and legal options are described in section 4). This can be changed in order to turn off, or down various analyses and transformations, inlining, specialisation, etc.

6.4.2 C code compilation

Once written and translated to C, a programs runtime behaviour may be monitored and influenced by runtime options. Some of the options are conflicting, and others cause performance problems. To avoid such problems a selection of compile time flags are used, and these determine which runtime flags will be available in the compiled program. To avoid conflicts, it is safest to link the compiled user program with a version of the runtime library compiled with the same set of options. The compile time options are (see the makefile for more details):

option	description
`TCK`	If defined, enables some run time consistency checking
`TCHK`	Enable generated code consistency checks at runtime
`SCHK`	Enable spine stack bounds checking
`CHK`	Enable more checks
`BOXITY_CHECK`	Check boxity of arguments after each function call
`ALLCHECKS`	All of the above
`PRIV_CHECK`	Zero private applications on first (and hopefully only) eval
`GC_STATS`	print out basic GC statistics
`STATS`	generate function call statistics
`TRACEABLE`	Enable runtime tracing
`MSSTATS`	Generate Store use graphs (needs fixing)
`ZEROONFREE`	Zero cells when known to be free
`NOGC`	No cell reuse. Allocate cells from one large contiguous area
`MALLOCGC`	Uses a garbage collector based on malloc that inspects the discovers live cells by inspecting the C stack
`REFCOUNT`	Use reference counting garbage collection

6.4.3 Runtime options

The following is a superset of the options available at runtime. The actual set of options available is governed by the flags provided at compile time (see above).

option	description
`-h`	Print help text showing available options
`-v`	Toggle statistic reporting
`-a`	Force all suspensions to be built using AP chains instead of vector apply nodes
`-r nnnn`	Main argument form
`-i`	Print the input caf only
`-s <file>`	Send call frequency statistics to `<file>`. `-s -` sends the statistics to stdout
`-P`	Output statistics in a form suitable for comparison with SASL statistics
`-d`	Display initialised data before commencing execution
`-f`	Sort statistics by frequency, rather than by name (default)
`-u`	unbuffered I/O
`-U`	Force update after each reduction. Used to monitor the success of privacy analysis
`-A`	Zero private APP cells after use
`-z`	Zero cells when collected
`-t <file>`	Output trace information to `<file>`. `-t -` sends output to stdout
`-L n`	Enable tracing after n calls to reduce
`-l n`	Generate traces with a level of detail no greater than n. Larger values of n cause an increase in output volume as argument data structures are displayed in more detail
`-H n`	Set default heap size to n bytes, when running with NOGC

6.4.4 Performance

For best performance and with partial garbage collection (via a malloc based collector), set `CPPFLAGS` to include `QUICKGC` (default for `CPPFLAGS` is `SMALLFLAGS`). `SMALLFLAGS` can be much slower, because much checking is performed at runtime, and no garbage collection is done. `SMALLGC` is similar in performance to `SMALLFLAGS`, but has malloc based collection. `SMALLGC` has proved to be reliable and the collector has not caused any problems to date.

6.4.5 Problems

A large amount of temporary space is required by the C compiler when compiling large programs. Under SunOS, tmpfs(4s) proved most useful in alleviating this.

7 Inside the production runtime system

This section describes the interface and features of the FAST production runtime system and code generator. The input is a series of macro calls generated by the compiler proper. These macro calls are placed in two files with `.cde` and `.hdr` extensions. The `.hdr` file contains data definitions and forward function declarations, whereas the `.cde` file contains the actual function bodies. These two files consist of unordered statements with sequencing keys, so they are first sorted, stripped of sequencing information and then translated to C code, compiled,

and linked with the runtime library. The end result is an executable binary. The macro calls examined in this section are stripped of sequencing information for reasons of clarity.

The production runtime system was originally intended to be a vehicle for testing and gathering statistics from the code generated by the FAST compiler. Currently it is the only execution vehicle for Haskell- compiled code. Some time has been spent profiling the runtime to eliminate major performance problems, and most, if not all statistics gathering and consistency checking calls may be declared as null macros by selecting the appropriate compile time flags. The facilities provided by the runtime system may be categorised as follows:

- boxed and unboxed data representations,
- static data declaration and initialisation,
- dynamic consistency checking,
- support for delayed evaluation,
- primitive functions (eg ADD, SUB),
- dynamic store allocation and reclamation,
- CAFs,
- statistics gathering,
- internals.

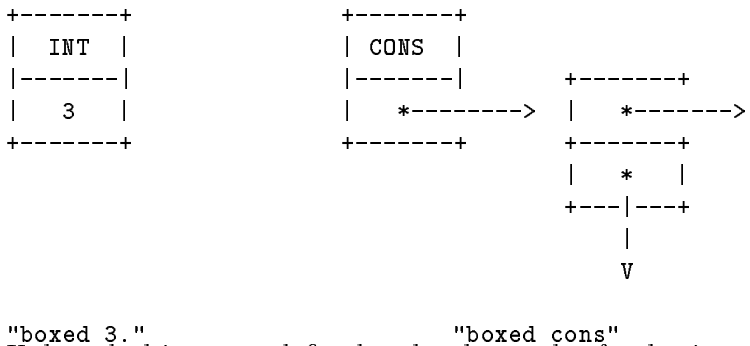
Each of these items will now be discussed in some detail

7.1 Boxed and unboxed data representations

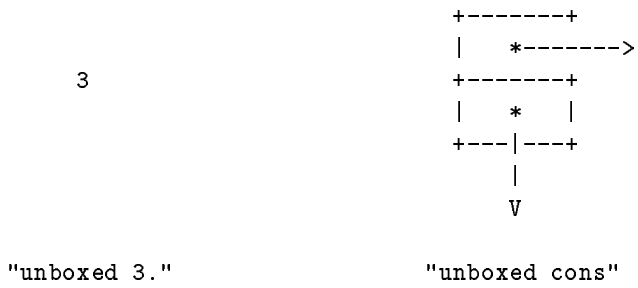
The notion of boxing is supported so that the runtime system may exploit the differences between boxed and unboxed data, resulting in a more efficient implementation. A boxed object may be a suspension, or a data item stored in the heap. An unboxed object must always be a HNF, and may or may not be stored in the heap. For an object to be shared, it must reside in the heap.

In this implementation of the runtime system, all unboxed data except cons cells may be destructively updated. Unboxed cons cells are stored in the heap, and all other unboxed data is passed as arguments, or stored in local variables. The destructive update restriction is applied to cons cells because they may be shared through the heap. Unboxed cons cells and boxed objects are currently passed by reference. Cons cells could be passed by value, allowing destructive updates within a procedure. The remaining unboxed objects are passed by value, and are never stored in the heap.

A boxed number is stored as a heap cell containing the machine representation of that number. A boxed cons cell is a heap cell containing a pointer to a heap cell which contains the head and tail of the cons cell:



Unboxed objects are defined to be the result of unboxing a boxed object. So an unboxed number is the machine representation of that number. An unboxed cons is a pointer to a heap cell which contains the head and tail of the cons cell:



7.2 Boxity, reducity, privacy and type

The boxity, reducity and privacy of an object are specified using a single digit for each property. Each digit may be interpreted as:

	boxity	reducity	privacy
0	boxed	unreduced	shared
1	unboxed	reduced	private

When referring to the form of an object, the boxity is specified followed by the reducity. [Here, the term "form" relates to the physical representation of an object, as opposed to privacy, which reflects how an object is used.] So, `_00` is boxed, unreduced; `_01` is boxed and reduced, and `_11` is unboxed reduced.

Throughout the generated code, identifiers are usually formed by taking the identifier name and adding a prefix which consists of the form of the object, with leading and trailing underscores to act as separators. Optionally, privacy and type information may be added as a suffix. In this example the suffix consists of an underscore followed by the type and privacy, eg: `_I_0` or perhaps `_I_1`. Thus, the variable name `_01_a_I_1` would suggest that `a` is represented as a boxed, reduced integer. A trailing `_1` suggests that `a` occurs in private contexts only.

FAST compiler input, be it basic or advanced Intermediate, places no typing restrictions on its input. Where the type of an object can be inferred from context and used to improve generated code quality, an attempt is made to do this. Such information is encoded in identifier names, as above, using one or more upper case letters from the following table:

type Identifier	object type
`A`	Array
`B`	Boolean
`C`	Character
`D`	Double precision floating point
`E`	Error, failed to infer type, so always boxed
`F`	Single precision floating point
`I`	Integer
`L`	List
`R`	Array descriptor, see Haskell manual
`S`	Array association, see Haskell manual
`T`	Any Type
`X`	Complex number

Thus `LLI` represents the type list of list of integers and `LSX` represents a list of associations with complex numbers.

It is sometimes necessary to convert objects during evaluation, as the form of an argument may not match that expected by a given function. In such cases, the argument must be converted to the correct form. For an object to be the correct form, the boxity must match the required boxity exactly, and the reducity must be greater or equal to (but not less than) that required. The compiler guarantees that the only operation needed to bring an object to the correct form is one of:

procedure name	operation
`_11_box_red_01`	unboxing
`_01_box_red_00`	reducing
`_11_box_red_00`	reducing then unboxing

The compiler's analysis is such that the generated code never needs to make an explicit call to box.

7.3 Static data declaration and initialisation

All data, whether function arguments, results, constants, or static data, is passed in a data type called Field. Because of this, static data, typically numbers, lists and characters, are associated with an identifier introduced by a declaration in the `.hdr` file. Due to the adoption of the Field type, it is not permitted to embed constants directly in the program text. Macros and procedures are used to place data into, and remove it from Fields. The macros and procedures are named thus:

```
<names> ::= get_<tname> | put_<tname>
<tname> ::= int | bool | num | ptr | fun | char | vec | file
```

eg:

```
put_fun(_01_ADD)
```

Returns an object of type Field, which contains the address of the function `_01_ADD`. In the `.hdr` file generated by the compiler proper,

```
hdr_numb(_11_117_D,1.0_D)
hdr_numb(_01_117_D,1.0_D)
```

will define both a boxed and an unboxed version of the constant 1.0. These calls will be used to define two items of type Field, `_11_117_D` and `_01_117_D`.

The use of this abstract data type allowed an early version of the runtime system to use C union structures to store floating point, pointer or integers in the same 32-bit location. The current implementation of the Field type no longer uses unions, but the interface remains unchanged.

7.4 Static lists and static suspensions

Static lists are introduced by a two argument `hdr_cons` macro. The first argument is an identifier, and the second is a list expression. The list expression may be either a straight forward cons operation:

```
hdr_cons(_01_249_LI,
        _01_CONS(_01_192_I,_01_244_LI))
```

or a more complicated expression involving a statically created suspension of `CONS`:

```
hdr_cons(_01_249_LI,
        vap(3,_01_CONS_0001T_0001LT_0_2,_01_192_I,_01_244_LI))
```

Because the `_01_CONS` primitive is lazy in both arguments, the code generator emits a static CONS cell for both of the above static data definitions. Static lists and circular structures may be built up from a number of such macro calls. Static suspensions of any function may be created, and the format of the arguments in the `vap` (Vector Apply) macro call are the same as those used in the `.cde` file described later.

7.5 Dynamic consistency checking

During program execution, three types of error conditions may occur:

1. logic errors in the runtime system,
2. errors due to incorrect compilation,
3. other errors, including type errors.

The runtime system attempts to differentiate between the class of errors, and alters the style of the error report accordingly. Errors attributed to 2 and 3 usually include an argument dump to aid debugging. All identifiers and parameters in the compiled code have names which indicate the current representation and the degree to which the object has been reduced. The code generation phase uses this information to insert consistency checks into the output C code. Such checks are one argument macro calls which take the object as their argument. A macro is defined for each of the possible representations and reducities. The macro naming is as follows:

```
form_br(_br_ident_p)
```

Where `b` and `r` are the boxity and reducity of the argument to be checked, and `p` is the privacy of the argument, which is not checked. Should a check fail, the execution is aborted, and a message is printed indicating the inconsistency. A debugger may then be used to find which assertion failed. The example below checks that the variable `_01_foo_1` is a boxed HNF:

```
form_01(_01_foo_1);
```

In addition to the above, the runtime system library contains calls to a family of two argument macros. These test their first argument, and abort printing their second argument should the first argument be false (ie. zero). These are:

procedure name	description
`chk(cond,str)`	called to check for internal inconsistencies in the runtime system
`tchk(cond,str)`	called to check for data inconsistencies at runtime, eg. tail applied to a nil list
`tc(obj,type,errmess)`	If "obj" is not of type "type", abort the program and print "errmess"

7.6 Preludes and want-have specifications

A prelude is a procedure which is invoked by the reducer when evaluating a suspension. Should the arguments in the suspension be in a form which is incompatible with that expected by the suspended function, then the prelude must convert the arguments to the correct form before calling the function. After the prelude has called the function, the prelude is responsible for updating the graph if necessary.

The form of the arguments expected by the suspended function, along with the form of those actually provided, is determined at compile time. This information, along with privacy information is communicated to the runtime system by means of a prelude specification. Currently, want-have specifications plus privacy are used for this. A want-have specification consists of the required (or want) boxity and reducity, followed by the current (have) boxity and reducity, and finally a type string, as described earlier.

A prelude specification consists of a want-have specifications, one for each argument position, and a privacy specification for the complete suspension. Each component of the specification is separated by an underscore. For example:

```
_0100I_1100I_1101I_0000LI_1
```

Is a prelude specification for a four argument function. The trailing `_1` indicates that it is used in a private (non shared) context, allowing updates to be safely suppressed. By taking the text of a want-have specification, stripping the type string, and inserting `_box_red_` between the two pairs of digits, we have built the name of an appropriate conversion procedure. These are the argument conversions implied by the above specification:

want-have	argument position	conversion procedure	description
0100I	1	_01_box_red_00	reduce
1100I	2	_11_box_red_00	reduce then unbox
1101I	3	_11_box_red_00	unbox
0000LI	4	_00_box_red_00	a No-Operation

7.7 Support for delayed evaluation

A mechanism is provided which allows the compiler to form a suspension for a computation, and to pass this suspension as an argument to other functions, possibly binding in further arguments. Eventually, the suspension may be unwound, yielding a HNF.

The mechanism used to create and evaluate suspensions uses a combination of application nodes and vector apply nodes with support for boxity and reducity considerations. This means that an application of a function can be implemented by a vector application node (VAP) or a chain of application nodes (APP chain). In the runtime system, a VAP is similar to an application cell, but the right pointer of the VAP cell points at an array of arguments, instead of just one. By having an indirection to the variable sized argument array, we avoid wasting the extra space associated with the argument vector when updating a VAP with a boxed result. We can also reclaim the argument vector separately, after we have extracted the arguments, before we update the VAP cell.

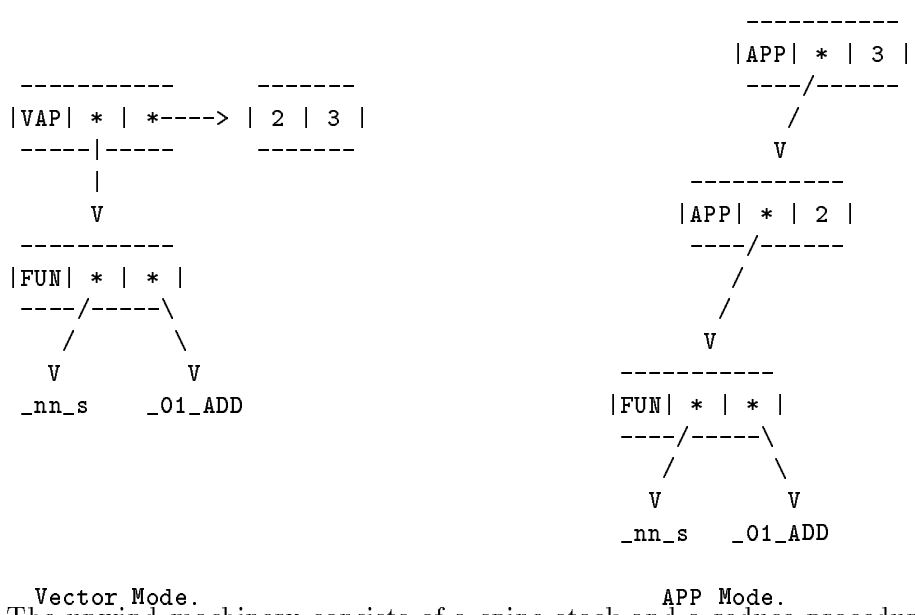
In the `.cde` file,

```
assign(_0X_173_LT,vap(3,_01_APPEND_1100LT_0000LT_0_2,
                       _0X_163_LT,_00__u2_LT_1))
```

creates a vector application cell at runtime, assigning it to the identifier `_0X_173_LT`. The `vap` macro takes a count indicating how many parameters follow, the name of a FUN cell, and arguments to the suspended function. Should there be insufficient arguments to make a total application, code to generate an application cell chain will be emitted.

7.7.1 An example: spines for the suspension of "ADD 2 3"

The following diagram illustrates the runtime representation, `_nn_s` is the address of the prelude procedure for `_01_ADD`, and `_01_ADD` is the address of the version of the addition function that will deliver an unboxed result.



The unwind machinery consists of a spine stack and a reduce procedure. Operations on the spine stack are as follows:

procedure	description
`push(i)`	Push the item `i` onto the stack
`i=pop()`	Pop the top of the stack into `i`
`new_frame()`	Creates a new, empty frame on top of the spine stack
`delete_frame()`	Removes the top frame from the spine stack. The top frame must be emptied using pop before a call to `delete_frame()` is made
`s_depth()`	Returns the number of arguments in the current stack frame

The spine stack data type is implemented using a static array, and new frames are marked with an array of boolean flags, one for each stack entry. The flag corresponding to the first slot in a new frame is set to mark the start of that frame.

The reduce procedure is only ever called by the two conversion procedures `_01_box_red_00` and `_11_box_red_00`, described previously. The argument to reduce is always boxed, and may or may not be a HNF. The reduce procedure consists of 2 states, an entry state s1, and a final state s2. On entry to reduce, state s1 examines the cell, and handles all operations except the unwinding of application chains, which is handled by state s2. The following table contains a case analysis of the behaviour of state s1:

node type	action
CAF	Reduce the CAF, returning the result
VAP	Apply function to args, return the result
APP	start unwind, and go to state 2
Box	return argument unchanged
Fun	return argument unchanged
<anything else, eg. unboxed cons>	Stop with error message

Reduction of a CAF cell takes place by pushing the CAF cell onto the spine stack, and calling the CAF prelude. The CAF prelude pops the cell, and calls the CAF procedure stored in the cell. Finally, the CAF prelude updates the cell with the return value from the CAF procedure. CAF cells are always updated.

The evaluation of a VAP is almost identical to that of a CAF. However, in this case, the VAP prelude also removes the arguments from the VAP argument vector, converting them to the correct form where necessary. The procedure is then applied to these arguments. Finally, the VAP prelude performs the update operation if required.

An APP node in state s1 causes reduce to call `new_frame()`, push the argument spine, and then jump to state s2 with the left child (`fst`) of the APP node as the new spine. State s2 must now complete the unwind. Boxed objects and FUN nodes cannot be further evaluated, and are returned unchanged.

When in s2, on top of the spine stack will be a frame containing at least one argument. Actions in state s2 are as follows:

node type	action
CAF	Recursively reduce the CAF, continue in s2
VAP	Evaluate VAP, continue in s2
Application	push spine and go to state 2 with left(spine)
Fun	push spine, call prelude
<anything else>	Stop with error message

7.8 Evaluating a vector apply suspension

Reduce finds the VAP node, and extracts the prelude from the spine thus:

```
prel = get_fun(fst(fst(spine)));
```

Next, reduce pushes a pointer to the VAP node onto the spine stack, and calls the prelude with `VAP_MODE` as argument:

```
tmp = (*prel)(VAP_MODE);
```

The prelude now examines its mode argument, and starts processing the VAP. First it pops the pointer to the VAP,

```
res = pop();
```

then, it extracts the suspended procedure `f`, and argument vector `vec`:

```
f   = get_fun(snd(fst(res)));
vec = get_vec(snd(res));
```

Now the arguments must be extracted from the argument vector, performing conversions where necessary (No conversions are needed in this example):

```
a1 = read_vap(vec,0);
a2 = read_vap(vec,1);
```

The suspended procedure may now be called directly, because the runtime system only uses VAP cells when all the arguments are present:

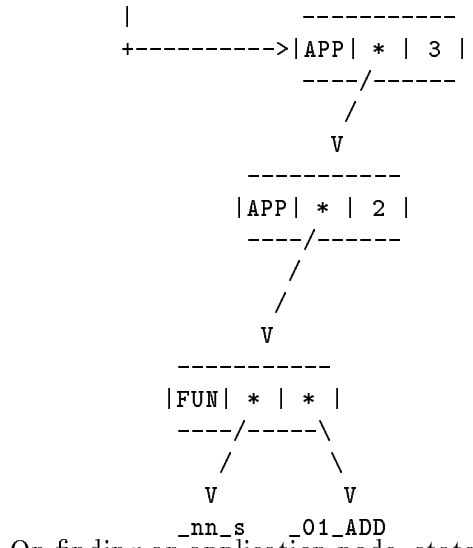
```
ans = (f)(a1,a2);
```

As soon as the function pointed at by `f` returns, the VAP cell can be updated:

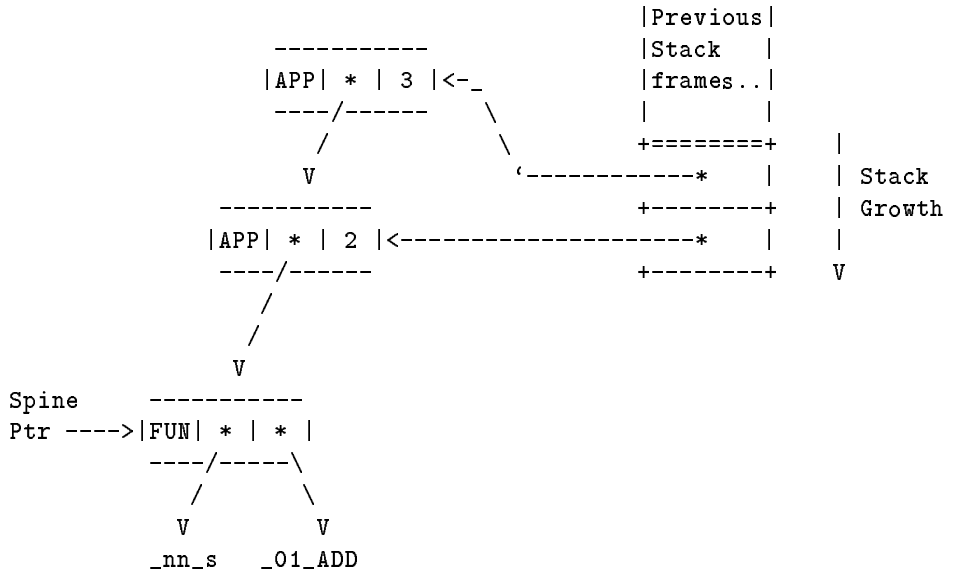
```
update(res,ans);
```

7.9 Evaluating an application chain

Reduce is called with a pointer to the top of the application chain thus:



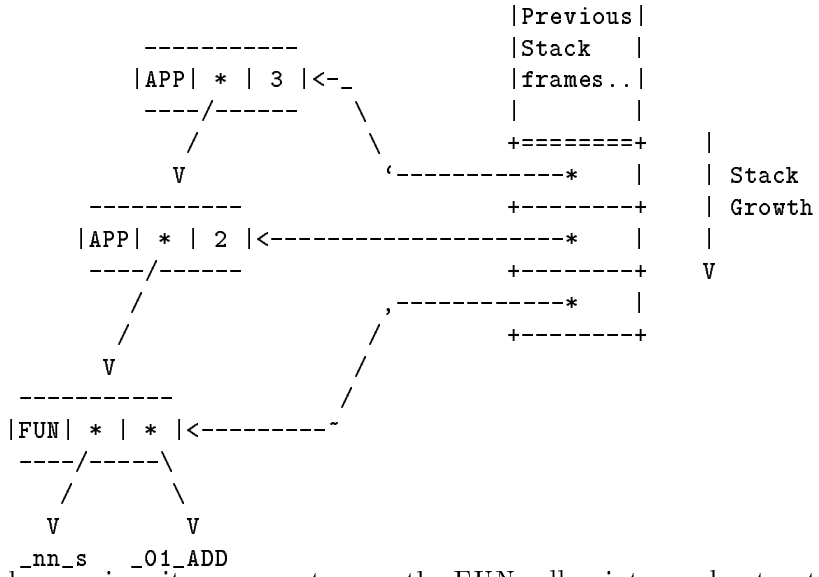
On finding an application node, state s1 of the reduce procedure creates a new stack frame and pushes a pointer to the application node. The spine pointer in reduce is set to the left child of the APP node, and control flows into State s2. State s2 finds another application node, pushes a pointer to it, and advances the spine pointer to the left child of the application node, and loops back to s2, leaving the stack as follows:



In state s2 a FUN cell is found, and reduce compares the arity of the procedure in the cell to `s_depth()`. (A FUN cell also contains the arity of the suspended procedure. Further details may be found in the Internals section, below.) The test returns TRUE, sufficient arguments are present, so reduce pushes a pointer to the FUN cell, and calls the prelude procedure `_nn_s`:

```
tmp = (*prel)(UNW_MODE);
```

The prelude finds the stack in the following state:



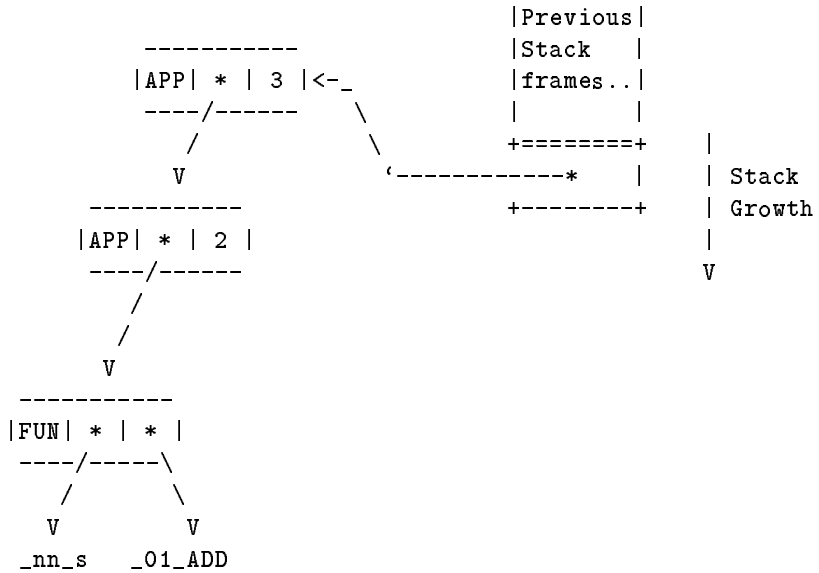
The prelude examines its argument, pops the FUN cell pointer and extracts the procedure pointer from the cell in one operation:

```
f = (Cfunp) get_fun(snd(pop()));
```

Next, the prelude pops off a pointer to the first argument:

```
a1 = snd(pop());
```

Leaving the spine stack as:



The prelude now pops off a pointer to the root of the application, storing a copy in the variable `res`, in preparation for the update:

```
res = pop();
```

The spine stack is now empty, and `delete_frame()` is called to remove the empty frame. The prelude takes the final argument out of the same application cell:

```
a2 = snd(res);
```

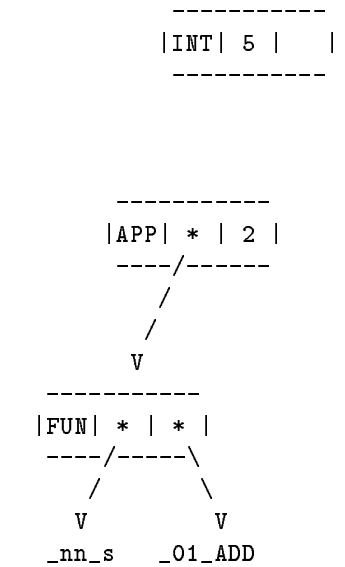
and calls the procedure:

```
ans = (f)(a1,a2);
```

Finally, the root of the computation can be updated with this result:

```
update(res,ans);
```

The suspension has now been transformed into:



with a boxed value 5 copied onto the root of the original spine.

7.10 Suspension forming operations

Assuming that we wish to generate a suspended call to a function named `zap`, which takes three parameters, with forms reduced, boxed and boxed respectively, then the code at the macro call level, could appear in the `.hdr` file as:

```
hdr_prel(_0100_1111_1100_0)
```

This would generate a prelude procedure `_rnu` (this stands for reduce, no action, unbox) for a shared suspension of a three argument function. The generated code would look like:

```
Field
_rnu_s(mode)
int mode;
{
    Field ans,res,a1,a2,a3;
    Cfunp f; Vec vec;

    if(mode == UNW_MODE){       /* Unwind code */

        /* Extract ptr to C code for the suspended function. */
        f = (Cfunp) get_fun(snd(pop()));

        /* Pop spine stack and dereference APP node to get argument,
         * then reduce and unbox as required.
         */
        a1 = _01_box_red_00(snd(pop())); form_01(a1);
        a2 =  snd(pop()); form_XX(a2);

        /* The last arg is a special case, we must obtain a pointer
         * to the root of the redex for update.
         */
        res = pop();

        /* remove stone from stone stack */
        delete_frame();

        /* Now we can remove and prepare the last argument. */
        a3 = _11_box_red_00(snd(res)); form_11(a3);

    }else{                      /* Vector code */
        /* Pop TC_VAP node off spine */
        res = pop();

        /* Extract ptr to C code for the suspended function. */
        f = get_fun(snd(fst(res)));

        /* Set vec to point at the argument vector */
        vec = get_vec(snd(res));

        /* extract arguments, reducing and unboxing as required. */
        a1 = _01_box_red_00(read_vap(vec,0)); form_01(a1);
        a2 = read_vap(vec,1); form_XX(a2);
        a3 = _11_box_red_00(read_vap(vec,2)); form_11(a3);
    }

    /* Call the suspended function */
    ans = (f)(a1,a2,a3);

    /* update the VAP or root APP node with a copy of the result */
    update(res,ans);
    form_00(ans);
    return (res);
}
```

In the above code, function calls to `form_nn()`, check that the boxity and reducity of the argument matches `nn`.

The prelude above takes a "mode" argument, which indicates whether the suspension was constructed using a single VAP cell, or multiple APP cells. The mode argument to the prelude is passed by reduce, indicating that reduce found a VAP node (`mode=VAP_MODE`), or an application chain (`mode=UNW_MODE`). Enough information exists at compile time to statically determine the type of prelude required for each call site, making the mode argument redundant. The approach described above reduces the number of prelude functions required, trading this for a small increase in execution time.

The latest version of the runtime system deviates from the mechanism described above, in that no mode argument is used, and instead the prelude is split into two functions, one which unwinds application spines, and another which builds a C stack frame from a vector application cell. Thus each statically allocated FUN cell contains two prelude function pointers. To streamline suspension evaluation further still, the preludes have been specialised for each user function, making it unnecessary to load the user function address from the FUN cell before calling the user function. This allows some primitive functions to be macro in-lined in prelude bodies, resulting in significant savings for suspensions of functions like `ADD`. In addition, `gcc` is now able to in-line some user functions. The motivation for these changes came from the need to perform reference count garbage collection on unevaluated suspensions. The solution adopted was to generate decrement procedures for each suspension, making use of the boxity and type information in prelude specifications.

7.11 Suspension activation

Suspensions are activated by being demanded. They may be demanded by an explicit conversion within a function body or a prelude. Such activation may be caused by a call to `_01_box_red_00`, if a boxed HNF is required (and similarly, `_11_box_red_00` for unboxed).

7.12 Dynamic store allocation

Currently, cells are not explicitly manipulated by the runtime system. Primitive functions cause cells to be allocated. Here are some examples.

function	number and type of cells claimed
box	1 (1 box)
`_01_ADD`	1 (1 box)
`_11_ADD`	0
`bind`	1 (1 application cell)
`_01_CONS`	2 (1 box and 1 cons cell)
`_11_CONS`	1 (1 cons cell)

Most boxed primitives need to allocate a cell to store their return value in. Notable exceptions here are the logical functions `_01_AND` and `_01_OR`, which return either one of their arguments, or one of the static boxed constants TRUE and FALSE.

Store reuse is implemented using a reference count garbage collector. At the time of writing, the C-library procedures `malloc` and `free` are called to allocate and free cells. This is a stopgap measure which makes performance comparisons meaningless, due to the massive function call overheads that this temporary approach imposes.

7.12.1 Constant applicative forms

The circumstances which cause the FAST compiler to generate CAFs is of no concern here. Here we present the runtime systems view of CAFs. One view of a CAF is that it is a statically declared heap cell containing a suspension. It is the compiler's way of creating application spines cheaply at compile time. Instead of building a VAP node containing a function applied to all its arguments, a procedure may be generated which contains an eager call to a function. This assumes that all the arguments to the suspended function are available as constants or addresses at compile time.

A pointer to the CAF procedure is then stored in the CAF cell, and when the cell is demanded, the procedure may be called directly, avoiding both the complicated unwind phase, and the allocation of a VAP cell. Furthermore, sharing may still occur through the statically allocated CAF cell. The address of the CAF cell which will be updated with the result of the CAFs procedure is now a constant address known at compile time. This cell may be directly referenced in all procedures. CAF are only available in boxed form. A CAF is a box, and, like an application cell, when reduced, a CAF returns a boxed object. CAFs are treated as special case in the runtime system. They are allocated a special type of cell, and reduce contains code specially for CAF evaluation. CAF cells are not garbage collected.

7.13 Statistics gathering

Statistics gathering is provided in three levels. The least expensive, enabled by the `GC_STATS` compile time flag, prints the number of static and dynamic cell claims, and a breakdown of the cells claimed by type. The next level, enabled by `GC_STATS` and `STATS` flags together provides call frequencies for functions, distinguishing between lazy and eager call sites, and at slightly more expense.

Finally, the `MSSTATS` option uses a conservative mark-scan collector to generate lifetime and graph size information. To use this facility, `MSGC` should be defined, and `RECLAIM` must be omitted. Use of this option can slow down execution dramatically. Unfortunately, at the time of writing, this facility has been broken by the addition of the reference counting collector.

7.13.1 Lazy function calls

When statistics are requested (`-s` option), the lazy calls are output last, and may be distinguished by the text "lazy" which appears in-between the function name and the counter thus:

```
_01_atomcentre_f1    lazy  10
_01_atomcentre_f2    lazy  11
_01_atomcentre_f3    lazy  11
_01_atomcentre_f4    lazy  16
_01_atomcentre_f5    lazy  16
_01_atomcentre_f6    lazy  16
```

7.13.2 Eager function calls

For each procedure definition, a variable is allocated to count invocations. This variable is incremented on entry to each procedure. It is incremented irrespective of whether it is called

from a lazy or eager context. This counter indicates the total number of calls. The number of eager calls may be determined by subtracting the Lazy function call total (see above).

```
01_atomcentre_f1      11
01_atomcentre_f2      12
01_atomcentre_f3      12
01_atomcentre_f4      17
01_atomcentre_f5      17
01_atomcentre_f6      17
```

This facility can be disabled by a compile time option. In order to facilitate the gathering of data between separately compiled modules, the addresses of all the counters are stored alongside the function names in a centrally managed table. The first time a counter is incremented, its record is inserted into the table. Subsequent calls simply increment the counter.

7.13.3 Cell claims

After successfully completing evaluation, programs compiled with `GC_STATS` defined will output counts indicating the number of CONS, APP, VAP and BOX cells claimed.

7.13.4 Live graph size

By scanning all the accessible program graph after each cell claim, we can produce statistics indicating the number of live (non-garbage) heap cells of each type. After each scan, counts representing the number of occurrences of each type of cell are output to a file, followed by a newline character. Output might appear as below, minus the headings, which are added here for clarity.

	boxes	conses	VAPs
0	0	0	0
0	1	0	0
0	1	1	0
1	1	0	0

The figures represent the number of live cells of each type. Each line represents the number of live cells after the call to allocate, but before the cell is actually allocated.

From the above trace we can see that the first cell claimed was a box, the second a cons. The third cell claimed was an application cell, and one cons cell became garbage before this call to alloc. However, when cells die between claims, it is not always possible to determine what type of cell has died from the above information alone. Of the cells which are dynamically allocated, only application cells and VAP cells change their types during evaluation. CAF cells also change their type, but these cells are neither collected nor dynamically allocated. The trace shows the number of cells in the graph that currently have a particular type.

7.13.5 Cell lifetime

We define the age of a cell to be the number of cell claims that occurred during the cells lifetime. A matrix of counters is maintained, indexed by age and cell birth-type. When a

cell dies, either the appropriate counter is incremented, or an overflow counter is incremented should the age be excessively large. After the program has run to completion we output the lifetimes as follows:

For cell ages starting from 1, print the number of cells of each type that survived to this age, followed by a newline. The following is a sample of output, again with header added:

APPs	boxes	conses	VAPs
0	0	0	0
30	250	0	3
70	180	5	15

This shows that 250 boxed cells lived just one claim, 5 cons cells lived for two claims, etc. The type of the cell is noted on allocation and the new cell is stamped with the number of cells claimed so far. Although the type of the cell may change in future, as a result of update operations, a shadow write once version of the cell type is maintained elsewhere. So, the lifetime statistics for each cell type reflects the type of the cell at birth, and not at death.

7.13.6 Real time interval between cell claims

It is possible to generate a trace showing the real time interval between successive cell claims. Both cell lifetime and graph size statistics are inhibited. It is recommended that three runs of the program are used, and that the minimum of the time intervals in the three log files is used as a measure.

7.14 Calls to reduce

The `-s` option generates a wide range of counts, some of which indicate the type and frequency of arguments passed to the reducer (red). Reduce is currently implemented as a two state FSM. The first (entry) state s1, handles all cases except unwinds. When an APP node is met, s1 stacks a pointer to the first application node, and passes control to s2, which completes the unwind. So boxed constants, fun cells, CAFs, and VAPs may be processed by s1. CAFs applied to arguments form an unwind chain, which is also handled by s2. Also, the number of calls to the reducer equals the number of cells processed in s1.

Here is a sample of some reduce statistics:

```
red_app                 1454
red_depth_lt            309
red_depth_eq            176
red_fun                 485
red_s1_app              485
red_s1_boxed            2571
red_s1_CAF              6
red_s1_vap              1430
 ....
red                     4492
```

The `red_depth_lt` count indicates the number of times reduce was called with a partial application. The `red_depth_eq` count indicates the number of spines which contained exactly the correct number of arguments. The `red_depth_gt` count indicates the number of times a

function is applied to too many arguments. It does not appear above, indicating that this did not happen in the above run. `red_s1_boxed` indicates the number of times reduce is called on a boxed number. `red_s1_vap` indicates the number of VAPs reduced. `red_s1_app` indicates the number of APP nodes met in state s1, while `red_app` indicates the number of APP nodes met subsequently.

7.14.1 Updates

If any updates have been performed, an indication of how many will appear thus:

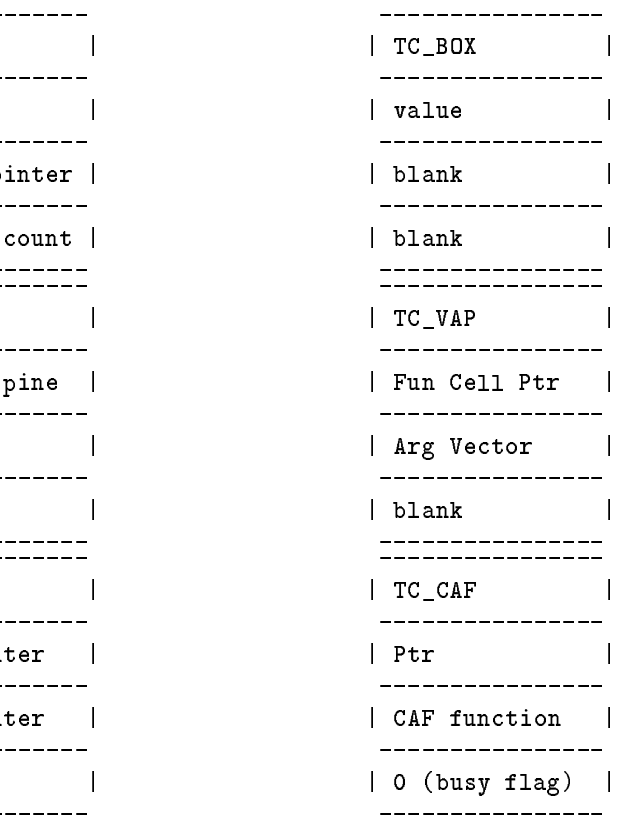
```
update                420
```

This figure represents the number of updates performed on VAP and APP cells. CAF cells are not dynamically allocated, and the cells are always updated by the CAF prelude. CAF updates are indicated thus:

```
p_CAF                 6
```

7.15 Cell layout

The following section details the cell format used inside the runtime system. All cells are preceded by a single reference count, which is left out for simplicity. Cells are used as follows:



The argument count of FUN cells contains the arity of the procedure referenced by the "C code pointer" field. FUN cells and CAF cells are both allocated and initialised before evaluation starts. The argument count field in CAF cells is used as a busy flag. It is set

to 1 when CAF evaluation commences. Should an attempt be made to evaluate a CAF cell containing a 1 in this field, reduce will halt with an error message, indicating that an erroneously circular definition has been found.

8 Concluding remarks

The FAST system is still under development, in particular where it concerns the use of parallelism. However we would like to encourage the use of the system so that improvements can be made based on the experience gained by the users.

There are bugs in the system that we want to know about, so please let us know (by e-mail) if you find a problem.

In this user's guide we have tried to describe the aspects of the system that are important to the users. We have not described everything that we should. In particular there is no description of the following:

- compile time and runtime error messages,
- a formal specification of the compiler passes (the compilation schemes),
- a list of what has been implemented in what version of the compiler and runtime system,
- a standard library of functions (we use the Miranda standard environment for this),
- how to spark and exploit parallel evaluation,
- how to improve the efficiency of programs.

9 Acknowledgements

We thank Stuart Bell for his contribution to the implementation of the FAST compiler [3]. Koen Langendoen and Stuart Bell made useful comments to a draft version.

References

[1] L. Augustsson and T. Johnsson. The Chalmers lazy-ML compiler. *The computer journal*, 32(2):127–141, Apr 1989.

[2] L. Augustsson and T. Johnsson. Lazy ML user's manual. Programming methodology group report, Dept. of Comp. Sci, Chalmers Univ. of Technology, Göteborg, Sweden, 1990.

[3] S. K. Bell. Extensions to the intermediate stage of the FAST Haskell compiler. Master's thesis, Dept. of Electr. and Comp. Sci, Univ. of Southampton, England, May 1991.

[4] A. Bloss. Update analysis and the efficient implementation of functional aggregates. In J. Stoy, editor, *4th Functional programming languages and computer architecture*, pages 26–38, London, England, Sep 1989. ACM.

[5] S. Cox, S.-Y. Huang, P. H. J. Kelly, J. J. Liu, and F. Taylor. An implementation of static functional process networks. In D. Etiemble and J.-C. Syre, editors, *Parallel architectures and languages Europe (PARLE), LNCS 605*, pages 497–512, Paris, France, Jun 1992. Springer-Verlag.

[6] P. H. Hartel, H. W. Glaser, and J. M. Wild. Compilation of functional languages using flow graph analysis. Technical report CSTR 91-03, Dept. of Electr. and Comp. Sci, Univ. of Southampton, England, Jan 1991.

[7] P. H. Hartel, H. W. Glaser, and J. M. Wild. On the benefits of different analyses in the compilation of functional languages. In H. W. Glaser and P. H. Hartel, editors, *Implementation of functional languages on parallel architectures*, pages 123–145, Southampton, England, Jun 1991. CSTR 91-07, Dept. of Electr. and Comp. Sci, Univ. of Southampton, England.

[8] P. H. Hartel and K. G. Langendoen. Benchmarking implementations of lazy functional languages. Technical report CS-92-XX, Dept. of Comp. Sys, Univ. of Amsterdam, Dec 1992.

[9] P. H. Hartel and W. G. Vree. Arrays in a lazy functional language – a case study: the fast Fourier transform. Technical report CS-92-02, Dept. of Comp. Sys, Univ. of Amsterdam, Presented at ATABLE-92, Montréal, Canada, Jun 1992.

[10] P. H. J. Kelly. *Functional programming for loosely-coupled multiprocessors.* Pitman publishing, London, England, 1989.

[11] K. G. Langendoen and P. H. Hartel. FCG: a code generator for lazy functional languages. In U. Kastens and P. Pfahler, editors, *Compiler construction (CC), LNCS 641*, pages 278–296, Paderborn, Germany, Oct 1992. Springer-Verlag.

[12] K. G. Langendoen, L. O. Hertzberger, and W. G. Vree. Design and performance modelling of WYBERT; parallel reduction on shared memory. In *Abstract machine models for highly parallel computers, Vol. II*, pages 17–22, Leeds, England, Mar 1991. School of Computer Studies, Univ. of Leeds.

[13] E. G. J. M. H. Nöcker, J. E. W. Smetsers, M. C. J. D. van Eekelen, and M. J. Plasmeijer. Concurrent Clean. In E. H. L. Aarts, J. van Leeuwen, and M. Rem, editors, *Parallel architectures and languages Europe (PARLE), LNCS 505/506*, pages 202–220, Veldhoven, The Netherlands, Jun 1991. Springer-Verlag.

[14] ed. P. Hudak, ed. S. L. Peyton Jones, and ed. P. L. Wadler. Report on the programming language Haskell - a non-strict purely functional language, version 1.2. *SIGPLAN notices*, 27(5):1–162, May 1992.

[15] D. A. Turner. A new implementation technique for applicative languages. *Software—practice and experience*, 9(1):31–49, Jan 1979.

[16] D. A. Turner. SASL language manual. Technical report, Computing Laboratory, Univ. of Kent at Canterbury, Aug 1979.

[17] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In J.-P. Jouannaud, editor, *2nd Functional programming languages and computer architecture, LNCS 201*, pages 1–16, Nancy, France, Sep 1985. Springer-Verlag.

[18] D. A. Turner. *Miranda system manual.* Research Software Ltd, 23 St Augustines Road, Canterbury, Kent CT1 1XP, England, Apr 1990.

[19] M. C. J. D. van Eekelen, H. Huitema, E. G. J. M. H. Nöcker, J. E. W. Smetsers, and M. J. Plasmeijer. Concurent Clean language manual - version 0.8. Technical report 92-18, Dept. of Comp. Sci, Univ. of Nijmegen, The Netherlands, Aug 1992.

[20] W. G. Vree and P. H. Hartel. Fixed point computation for parallelism. Technical report CS-92-07, Dept. of Comp. Sys, Univ. of Amsterdam, Jul 1992.

[21] J. M. Wild, H. W. Glaser, and P. H. Hartel. Statistics on storage management in a lazy functional language implementation. In K. Boyanov, editor, *3rd Parallel and distributed processing*, Sofia, Bulgaria, Apr 1991. North Holland.