ESPRIT PROJECT 6062

Description and Evaluation of the Initial Dylan Producer and Runtime Support

Written by: Ian Piumarta
Harlequin Ltd.

Issued by: Jorgen Bundgaard
DDC International

Delivered by: Gianluigi Castelli
Project Director

The status of this document is "WORKING" if signed only by its author(s); it is "ISSUED" if signed by the Workpackage Manager; it is "DELIVERED" if signed by the Project Director.

Project deliverable id: TR4.2.2b-01

Document code: Harlequin GLUE4.2.2b - 01

Date of first issue: 1994-05-21
Date of last issue: 1994-05-21

Availability: Confidential

The contents of this document are confidential and subject to copyrights protection. Any infringement will be prosecuted by law.

omi/glue

CHANGE HISTORY

This is the first version.

Contents

1 Purpose **2**

2 Executive Summary **2**

3 Scope of the Initial Producer **2**

4 Design Goals **3**

5 Overview of the Dylan Producer **4**
5.1 Intermediate Representation 4
5.2 Library Organisation 5

6 Overview of the Generated TDF **6**
6.1 Declarations and Definitions 6
6.2 Accessing and Modifying State 7
6.3 Conditionals and Control Flow 7
6.4 Function Application 8
6.5 Literals 8

7 Future Directions and Requirements **9**
7.1 Calling Convention Abstraction 9
7.2 Internal and External Entry Points 10
7.3 Dynamic and Static Booting 11

8 Suitability of TDF for Language and Runtime **12**
8.1 Function Bodies and Conditionals 12
8.2 Problems with Dead Code 13
8.3 Missing SORTNAME 13

9 Performance **13**

10 Conclusions **14**

A The Benchmarks **15**
A.1 Dylan Source 15
A.2 Generated TDF 16
A.3 Generated C 20

References **24**

1 Purpose

The purpose of this document is to assess the suitability of TDF [FC92] as a target for compiled Dylan [App92]. In order to do this, the TDF generated by Harlequin's initial Dylan producer is described and an analysis made of the TDF produced.

This work was sponsored by the Commission of the European Communities.

2 Executive Summary

Dylan is an advanced, modern, dynamically-typed, object-oriented programming language. It contains features not found in most other languages — such as automatic memory management, and an exception mechanism. The language is still under development, although most of the design work that remains concerns the standard language libraries rather than the semantics of the language itself.

As part of their work with OMI/GLUE, Harlequin have undertaken to implement a Dylan producer. This will establish the adequacy of TDF for the implementation of modern, dynamically-typed, object-oriented languages, and also allow comparison of the run-time efficiency of a TDF implementation with native implementations based on compilers using the same front-end, intermediate representation and optimisation technologies.

This document presents Harlequin's initial Dylan producer, in terms of the TDF sequences corresponding to various constructs in Dylan source, and the initial implementation of a token library for Dylan. The Dylan token library must be designed carefully to ensure it supports different combinations of run-time activities (garbage collection strategies and calling conventions being two important examples) whilst keeping the externally visible set of tokens simple.[1]

More than simply a "porting technology", Harlequin see TDF as the ideal vehicle for experimentation with implementation techniques. The design of the Dylan producer emphasizes the possibilities inherent in TDF for rapid modification and evaluation of implementation techniques.

3 Scope of the Initial Producer

The initial producer is limited in scope for a number of reasons.

There has not yet been any development work on *integrated* garbage collection algorithms.[2] The initial Dylan producer uses a conservative garbage collector which is completely transparent to the Dylan token library, so no special token definitions are necessary to aid garbage collection. The places in which garbage collection strategy

[1] This is desirable for two reasons: a simple token library will both keep the compiler back-end relatively simple (by moving complexity out of the compiler and into the token library), and also provide for future extensibility without the need for extensive modification to the compiler (ideally only the definitions in the token library will change).

[2] By "integrated" we mean a garbage collector which relies upon compiler support for its operation. Such compiler support might be the inclusion of read or write barriers, context maps for each function, and so on.

impinges on the library token definitions will not be clear until we begin to support more integrated garbage collectors.

Similarly, the initial Dylan producer does not use advanced calling conventions. It uses a C-style calling convention employing only *caller* parameters and ignoring *callee* parameters entirely.[3] The places where calling conventions impinge on the library token definitions will not be clear until we start to support more elaborate calling conventions.[4]

The initial producer does not rely on any low-level mechanisms for dealing with tagged data. Nor does it have any library tokens which provide an abstraction around *boxed* values [Man93a].[5] The places in which support for tagged and/or transparent handling of boxed values impinges on the library token definitions will not be clear until support for these is added at a later date.

Some of the more interesting work that could be described directly by token expansions is instead pushed into the runtime support (written in C, and shared between the Dylan producer and a separate back-end which generates C); this is especially true of some of the work for accessing boxed values mentioned above. Consequently, any runtime activities that would impinge upon library token definitions in a production quality Dylan producer will not be clear in the initial producer.

4 Design Goals

The initial Dylan producer has a modest set of design goals. Its primary purpose is to facilitate an evaluation of the suitability of TDF as a target language for Dylan. What is important at this stage is to identify quickly if there are any serious omissions in TDF that will hamper even a naïve implementation of a Dylan producer.

A popular route to platform independent code generation is to target a language to C. This has been a parallel strategy adopted by Harlequin in their Dylan implementation efforts. A secondary design goal has therefore been to keep the compiler technology sufficiently close to an existing C back-end to facilitate an evaluation of the efficiency of the installed TDF executable with respect to an executable created via another platform independent representation.[6]

The final design goal at this stage, although of much less importance than the preceding two at the moment, is to provide a degree of algorithmic interchangeability in certain important subsystems of the implementation (for example, the garbage collector and the function calling mechanism). Although this has been borne in mind while

[3] A parameter is classified as caller or callee according to the party responsible for removing it from the stack (or some other cleanup operation) before or after the function returns.

[4] The use of callee parameters allows some important, if not fundamental, optimisations. These include *tail call elimination* in which the result of a function call is provided by a call to another function: the arguments to the original function are replaced *in-situ* by those for the new function and a tail-jump (rather than a call) executed to effect the argument passing, function call, and a return from the original function. Note that this is possible in implementations that use caller rather than callee parameters, providing care is taken that a tail-jump to a function requiring more caller parameters is never made.

[5] This is not *strictly* true: there are two tokens that provide insertion and extraction of values from (numbered) slots in a proper object. These tokens are equally well suited to extracting the value from a boxed object.

[6] At this stage there is one *advantage* in retaining the C-style calling convention for Dylan, namely that the existing C runtime system can be used practically unmodified for the Dylan producer.

developing the Dylan producer, it has not yet had much impact on the design since the initial producer uses a transparent garbage collector and supports only a C-style calling convention.

The abstraction of important subsystems such as function calling conventions, garbage collection techniques, and manipulation of tagged data, allows rapid experimentation with the implementation of these subsystems. TDF link and install times are significantly lower than compilation times for real programs, such as the Dylan compiler itself. Development and evaluation of new language implementation strategies can be hampered by the time taken to recompile the compiler, language test suites, and benchmarking programs. By moving as many of the details of the implementation into the Dylan token library, the "edit-compile-evaluate" cycle for experimentation with implementation techniques should require significantly less time: changes to token definitions will become effective as soon as the modified token library is recompiled, and the (already produced) test-suite and benchmark programs need only be reinstalled (with the modified token library) before the new implementation strategy can be evaluated.

5 Overview of the Dylan Producer

This section provides some context in which the rest of the document can be read.

5.1 Intermediate Representation

Traditionally, compilers use one or two intermediate representations for code. The first of these is usually a parse tree which is a structured representation of the derivation of the source program, according to productions in the grammar of the language in which it is written. This tree carries both syntactic and semantic information, although it is the latter that is important for code generation. Code can be generated directly from the parse tree by a variety of techniques, or a second intermediate form such as 3-address code can be generated before finally emitting the executable code.

Each representation is amenable to different kinds of analyses and optimisations; higher-level structured representations being good for *global* optimisations involving mutation and/or movement of code, and lower-level linear representations being good for *local* (or *peephole*) optimisations such as move chain and dead code elimination.

Harlequin's Dylan producer uses a single "medium-level" intermediate representation, which to some extent allows both global and local optimisations. This takes the form of a directed acyclic graph (DAG) composed of several layers, in which lower layers describe individual operations, and higher layers group the lower layers into structural units. This representation carries more semantic information, but destroys most of the syntactic information in the original source.

Flow of control is represented in the DAG as edges between nodes, where each node is a basic block.[7] The back-end can only emit a basic block as an entire unit:

[7] A basic block is a fragment of code that starts with a label and ends in a jump or a fork (a conditional jump to one of two labels). There are never any jumps into the middle of a basic block.

basic blocks can never be divided into individual statements for the purposes of code generation.[8]

5.2 Library Organisation

The TDF generated by the Dylan producer makes use of two external sources of definitions. The runtime environment is described by a set of tagdefs, one for each of the functions provided by the runtime system. In addition to these, a library of predefined tokens insulate the generated code from activities that are necessary for particular implementation subsystems, such as the garbage collector and calling conventions.

Any particular subsystem, the garbage collector for example, may have no single "correct" implementation since there may be many possible implementations, each having a different set of characteristics, with the best choice depending on factors such as the particular platform upon which the installed code will run or the nature of the application being compiled.[9] Depending on the particular garbage collection strategy employed it might be necessary to include extra operations and / or information in the generated code such as write-barriers, or context maps for the garbage collector to use when tracing the stack [Man93b].

This insulation is desirable since the compiler back-end is kept independent of changes in the particular strategies employed to solve any given runtime problem; thus a change in implementation strategy, at the TDF token level, will not require any changes to the compiler back-end.

This approach also facilitates rapid experimentation with different implementation strategies. Given several different garbage collector implementations, each described by a different set of TDF tokens sharing a common external protocol, comparisons can be made against each strategy by linking and installing a test suite or benchmark (which need be compiled only once) with each library.

To achieve some degree of independence between the compiler back-end, the various orthogonal axes along which implementation subsystems can be changed, and any underlying support for pervasive features such as tagged data, the organisation shown in figure 1 has been adopted for the token library. However, little of this structure is present in the library that accompanies the initial producer due to the simplicity of the initial garbage collector and calling conventions, and the lack of any tokenised support for optimised representations of data.

[8] This restriction is imposed by the part of the compiler "middle-end" that delivers information to the (target-specific) back-end.

[9] A program which manipulates data off-line, generating huge amounts of garbage in the process, will want to reduce the time spent collecting garbage to a minimum — regardless of any long pauses that may interrupt normal processing. On the other hand an interactive drawing package would be better with an *incremental* collector (one which collects a little garbage every time it allocates a little storage), devoting a larger percentage of the CPU to garbage collection, but without introducing any noticeable pauses into normal processing (which is essential in any interactive context). Other types of application might be best matched with one of the many other types of garbage collector, or in some cases it might be best to have no garbage collector at all.

Figure 1: organisation of the Dylan token library. The dylan token library has three levels. A *high-level library* which contains a small, platform- and implementation-neutral set of tokens visible to the compiler back-end. Below these are the *implementation libraries* containing tokens which implement the various subsystems (garbage collection, calling conventions, *et caetera*). The lowest level is any required *platform library* containing any platform-specific (or implementation-pervasive) tokens for activities such as the manipulation of tagged data.

6 Overview of the Generated TDF

This section presents an overview of the TDF generated by the initial Dylan producer. It is far from a complete description of the TDF generated by the Dylan producer, but will be sufficient to give an idea of the implementation of the important features of the language.

6.1 Declarations and Definitions

Almost all values in a Dylan program are *object pointers*,[10] therefore most variables (and function return values) have type "pointer to object". The Dylan library defines a shape called `pz` to represent an object pointer, which is used almost universally in declarations for variables (both global and local) and functions.

Since functions bodies are collections of basic blocks (section 5.1), with jumps always occurring to the start of a basic block, it is most natural to define them as a single LABELLED statement:[11]

[10] An object pointer is precisely that: a pointer to the structure representing an object (some combination of class, size, and the fields holding the state of the object would be typical). In dynamically typed object-oriented languages, objects are normally known by their object pointers: variables always contain object pointers, parameters are always object pointers, and so on. Tagging is often used to encode both the class and data for some objects directly within the object pointer (typically integers and characters are represented like this), in which case the object pointer is not strictly a pointer at all.

[11] All the examples of TDF in this document use the notation accepted by the TDF Notation Compiler.

```
(local make_id_tagdec z_emit__if - proc)

(local make_id_tagdef z_emit__if
        (make_proc pz
                pfpz - function
                pz - next_methods
                pz - z_stream
                ...arguments
        | - (sequence
                (variable var_acc RET__0 UNBOUND (sequence
                (variable var_acc ARG__0 UNBOUND (sequence
                ...local variables
                        (labelled block_intro LBL__0 LBL__1 | (goto block_intro)
                                (sequence ...basic block )
                                (sequence ...basic block )
                                (sequence ...basic block )
                        ))))))))
```

(The alert reader will, by now, be wondering what a "pfpz" is: it's a *p*ointer to a *f*unction returning a "pz"!)

6.2 Accessing and Modifying State

Variable accesses are either by reference (an *lvalue* in C parlance) or for their value (an *rvalue*). All accesses of this type are handled by Dylan library tokens which take the variable's TAG as an argument. An additional library token implements assignment:

```
(set (lval z_destination) (rval z_source))
```

These tokens have been introduced both as a means to slightly reduce the size of the generated TDF, and also as hooks onto which implementation-specific actions (such as the imposition of read or write barriers) can be hung.

6.3 Conditionals and Control Flow

Conditional statements (`if`, `case`, and so on) are all converted into a generic conditional construct which performs a test and branches to one of two labels depending on the result. Due to optimisations performed on the intermediate representation before TDF is emitted, one or other of these branches is often elided. Therefore it is necessary to have three tokens that perform condition branching, plus one other that performs an unconditional branch.

(goto *aLabel*)	*unconditional*
(goto_true *value aLabel*)	*branch to aLabel if value is true*
(goto_false *value aLabel*)	*branch to aLabel if value is false*
(goto_true_false *value tLabel fLabel*)	*branch to [tLabel fLabel] if value is [true false]*

The Dylan library also includes various tokens for constructing relational expressions, such as eq, neq, and so on, which are used mainly within expressions forming the *value* arguments to the goto tokens.

6.4 Function Application

Function application falls into two categories. *Primitive* functions[12] are applied via the APPLY_PROC token, with each primitive function being treated specially by the producer. There are many such primitive functions, and a discussion is beyond the scope of this document.[13]

The application of a *generic function* [Man93a] is a relatively complex task involving processing of arguments, construction of a list of *sorted applicable methods* [Man93a], and so on, before the intended destination method is called. This work is carried out by the runtime system and a handful of auxiliary functions (implemented in both C and Dylan) on behalf of the generic function, so the TDF generated in such cases simply calls the relevant procedure in the runtime system with information specifying the number of actual arguments, the object representing the generic function to be called, the arguments themselves, and so forth. For example:

```
(apply_proc pz
        (lval CALL2)              runtime function which applies a GF to 2 arguments
        (rval z_format)           the generic function object to be applied
        (rval z_stream_46_M_)     first actual argument
        (rval z_LIT__16))         second actual argument
```

6.5 Literals

All literals are initially defined as "unbound". Each Dylan source file has functions responsible for initialising these literals, in one of two ways. Simple literals are built by calling the appropriate (Dylan) constructor function. Larger and/or more complex literals are built by generating a static structure containing a "program" describing the literal, which is evaluated by a tiny stack-oriented interpreter to yield the appropriate value.

A handful of tokens are defined in the Dylan library to help with this process. For example, a literal list containing an integer, a character, and a string (maybe `'(42 "hi!" #\a)`) results in the following TDF:

```
(make_id_tagdec z_LIT__0 - pz)
(make_id_tagdef z_LIT__0 UNBOUND)

(local make_var_tagdec Fstr1 - (nof 4 (integer  char)))
(local make_var_tagdef Fstr1 (make_nof_int  char "hi!\000"))

(make_var_tagdec z_LIT__0__FASL - pz)
(make_var_tagdef z_LIT__0__FASL (make_compound pz_offset
        (FSpec 0 INTEGER (raw_integer 42))
        (FSpec 1 STRING (lval Fstr1))
        (FSpec 2 CHARACTER (raw_integer 97))
```

[12] Usually, a "primitive function" is one which supports the operation of a language but which is not (for reasons of efficiency or possibility) written in that language.

[13] Although not strictly the same, numeric operators are treated similarly to primitive functions and receive special treatment in the back-end; thus most of the available numeric operations in TDF can be used by the Dylan producer.

```
(FSpec 3 LIST (raw_integer 3))
(FSpec 4 FEND FEND)))
```

The variable z_LIT__0 will hold the value of the literal, z_LIT__0_FASL is a structure containing the "program" describing the literal, and variables such as Fstr1 are out-of-line strings used for creating symbols, keywords, floating point numbers and so on. This mechanism is part of the *dynamic bootstrap* mechanism, and such complex literals will soon be described as initialised structures when the *static bootstrap* comes into use (see section 7.3).

7 Future Directions and Requirements

As is probably apparent from the previous sections in this document, the most important future work is to provide callee parameters, an integrated garbage collector, and support for platform-specific hardware features (such as tagged data types). The long-term intention is to support several interchangeable algorithms and / or hardware characteristics along each of these axes, using abstractions at the TDF token level both to make them interoperable and also to present a single, unified token interface to the compiler back-end in the Dylan producer.

Although our experiences with TDF as a delivery vehicle for Dylan applications have thus far been promising, we do not envisage future developments to be completely free from compromises. This section presents briefly some of the predicted problems and areas of concern.

7.1 Calling Convention Abstraction

The initial Dylan producer remains firmly within a C-style calling convention, using caller parameters exclusively. This hides a potential compromise which may have to be made regarding function application.

In TDF, the apply_proc token is given all the information required (function, arguments, return type, and so on) to generate code to apply the function atomically (in a single token expansion). In future we will want to support more elaborate calling conventions, possibly by defining our own tokens as replacements for, but eventually making use of, apply_proc. The back-end then need not worry about the particular calling convention being used since such details are hidden behind the particular definition of the apply_proc replacement provided with any particular Dylan token library.

The ideal solution would be to allow LIST-valued arguments in token definitions. However, DRA have pointed out that this would necessitate constructor and accessor tokens to manipulate such arguments, and that such extensions could introduce algebraic problems into TDF itself. Given the present definition of TDF there are two possible solutions to this problem.

The Dylan language definition allows an implementation to "place a reasonable limit on the number of arguments that may be passed to any function" [App92], and Harlequin's Dylan implementations impose a limit of 255.[14] Since there is a finite

[14] On some architectures there are significant performance gains to be made if the limit is kept to a relatively small number which fits into a single byte.

upper limit on the number of parameters that may be passed to a function it is possible to define a token in the library for each possible arity of function call. This solution moves complexity out of the back end and into the library (which is, in general, the direction we want complexity to flow), but has the distinct disadvantage that a large number of tokens would be redefined each time the calling convention is modified (although in practice, once they were stable, modifications to the calling conventions would be a very rare).[15]

Alternatively it would be possible to have the back-end emit tokens before, during, and after an apply_proc. These tokens would mark, or wrap, important points in the function application. For example there would probably be tokens to mark the beginning and end of the call, and also wrapper tokens introduced to the apply_proc arguments to aid in the computation of appropriate proc_props and similar values. This solution has the advantage that the token library need only redefine a few tokens (which would have equivalents in the first solution) whenever a calling convention is modified. However there is also the disadvantage that at each function application in the generated TDF there will be many tokens that have no effect on the generated code whatsoever when the calling convention is simple.

This may seem like a trivial problem, but implementations of languages such as Dylan have a much broader conception of "calling convention" than do conventional languages such as C. For example, our calling convention for Dylan will soon include mechanisms such as method and inline caches to improve performance.[16] The TDF generated by the Dylan producer must therefore not only cater for differences in the caller/callee parameter conventions, but must also be flexible enough to cause code to be installed that implements none, one, or both of these types of cache.

7.2 Internal and External Entry Points

Object-oriented languages often provide inheritance as a mechanism for code reuse. Dylan follows in this tradition, providing inheritance through generic functions. A generic function has many implementations (called *methods*), the particular methods appropriate to any particular call being determined by the class of each of the actual arguments.

Dylan, in common with many dialects of Lisp, also allows optional parameters in function definitions. Corresponding optional arguments are not identified "positionally", but rather by keywords which prefix each optional argument in the sender and correspond to named parameters associated with the same keyword in the receiver's definition.[17] Also, Dylan supports so-called *rest arguments*: an indeterminate number of "anonymous" arguments which may be passed after all the required arguments. Rest arguments passed to a generic function are processed into a runtime data struc-

[15] In practice, were this to be the adopted approach, the token definitions would be generated programmatically.

[16] A *method cache* remembers which method was invoked for any given combination of generic function and set of argument types; such a cache could be global (hashing on the generic function as well as the argument types) or local to each generic function. An *inline cache* remembers the last set of argument types at each call site and which method was invoked as a result. The next call from that site can go straight to the appropriate methods if the arguments have identical types.

[17] Note that Common Lisp has two mechanisms which provide optional parameters: parameters identified by keyword (as in Dylan) and by position (with defaults if they are not present).

ture (such as a list or a vector) before being passed to the appropriate method as a single "regular" argument, corresponding to the named *rest parameter* in the function definition.

Our current Dylan implementation therefore has two *entry functions*[18] for any given generic function: an *external entry function* performs the additional processing required to fix the keyword and rest arguments, and an *internal entry function* which expects this processing to have been done already.

We will soon be moving to a calling convention that uses mixed caller and callee parameters. The callee parameters will be used for all of the "normal" arguments to functions (the required arguments, and any keyword and rest arguments, or the structures that have been built by an external entry function for them). Caller parameters will be used to build up implicit chains of information that typically has dynamic extent that mirrors the dynamic behaviour of the program itself.[19] Such information includes chains of context maps (used by certain kinds of garbage collector) and the *next methods* lists (used to call less specific methods in the same generic function, allowing a more specific method to be "wrapped" around a less specific one).

We wish to avoid wasting contexts between external and internal entry functions — the ideal situation is for an external entry function to perform a tail-jump to an internal entry function after any necessary argument processing (the implementation is free to modify the callee parameters in any way it chooses, since the function that performs the modifications is responsible for tidying up before returning). We will also want to pass a subset of the caller parameters from the external entry function to the internal entry function.

One way to arrange this is to have the internal entry function expect a subset of the caller parameters expected by the external entry function. By ensuring that the caller parameters of interest to the internal entry function appear first, and in the same order, to the external entry function it should be possible to tail-jump to the internal entry function. The internal entry function will be expecting a "prefix" of the caller parameters passed to the external entry function, and should ignore the extra arguments without the need for any special action on the part of the producer.

7.3 Dynamic and Static Booting

The initial Dylan producer uses a *dynamic bootstrap*, in which all literals are initially undefined. Code is produced for each source file which must be run to initialise all such literals before the compiled program can be used. After compilation a Dylan application must be run once for this initialisation process to happen, after which an image of the running program (with all literals initialised) is dumped. This dumped image becomes the final runnable application.[20]

Work is currently progressing towards a *static bootstrap* in which literals are described directly in the produced code, in exactly the same way that initialised aggre-

[18] Different *entry functions* are associated with different means of entering a given method. Each entry function has an associated *entry point* to which control is passed in order to enter the corresponding method.

[19] In certain situations this information will have indefinite scope, and appropriate action will be taken in these exceptional cases.

[20] This mechanism is identical to the way that most Lisp implementations initialise themselves.

gate structures in C would be described. This will place much heavier demands on TDF than does the present dynamic bootstrap, but no difficulties are envisaged given that TDF is capable of describing any static structure that a C program might require.

8 Suitability of TDF for Language and Runtime

Only a few specific problems were encountered, and were due to the particular intermediate representation used by the initial Dylan producer, rather than any gross deficiencies in TDF itself. Although the ideal solution in some of the following cases would be to modify either the intermediate representation or the way in which the compiler optimisations modify it, it should be noted that the Dylan producer is one particular back-end for a compiler that supports back-ends for many different architectures. Since all the initial phases of compilation are shared between all these back-ends, it is undesirable to modify any aspect of the initial phases of the compiler since this will cause a significant amount of work to be required in updating all the back-ends to accommodate those changes.

8.1 Function Bodies and Conditionals

Function bodies are always kept as a DAG of basic blocks (as described in section 5.1). When generating code for a function, code should be emitted for each of the basic blocks in an order that maximises the number of jumps which immediately precede their destination, thus permitting many of these jumps to be elided. However, it is difficult to elide any of these labels without considerable extra work in the producer[21] since these might still be valid destinations for loop entry points, or for alternative branches of conditional constructs. This poses a problem for a Dylan producer, since it is not possible in TDF to make uncontrolled use of labels and gotos.

The solution to this problem, in the initial producer at least, is to generate the entire body of a function as a labelled construct, with each of the basic blocks of the function appearing as sequence associated with its label. This organisation is also intimately connected with the way conditionals are treated.

The initial Dylan producer does not make use of the tokens provided for conditional execution (conditional and integer_test), because of the way the back-end handles blocks. The entire body of a function is represented as a collection of labelled basic blocks, each of which is emitted as a sequence within a large labelled construct. Any conditionals within this structure are converted into explicit tests and jumps in the intermediate representation before the back-end has a chance to emit any specialised conditional constructs.[22] For example, the trivial Dylan expression

```
if (test)
     consequent;
else
     alternate;
end;
```

[21] We expect the installer to perform the equivalent analysis.

[22] This is due to the heritage of the intermediate code, which was originally designed to make the generation of machine code easy. One can see that the behaviour mentioned above is much more sensible when the target language is machine code than when it is a more abstract language such as TDF.

results in the following TDF:

```
(labelled block_intro LBL__0 LBL__1 | (goto block_intro)
        (sequence # block_intro:
                (if (neq (rval z_test) (rval z__pc_false) )
                        (goto LBL__1))
                (assign RET__0 (rval z_alternate) )
                (goto LBL__0))
        (sequence # LBL__0:
                (return (rval RET__0)))
        (sequence # LBL__1:
                (assign RET__0 (rval z_consequent) )
                (goto LBL__0)))
```

The initial producer relies on the installer to perform analysis and optimisation of labelled constructs like these.

8.2 Problems with Dead Code

Some basic blocks within the intermediate representation may generate no code at all. It is impossible to simply ignore such blocks since they will already have been given labels (in the labelled construct which surrounds a function body). The solution adopted in the initial producer is to emit a "no-op" at the start of every sequence:

```
(sequence (make_null_ptr (alignment top)))
```

(which relies on the TDF installer to to elide these in every case).

8.3 Missing SORTNAME

Although only a very minor problem, it was noticed that there is no sortname for tdfstring (analogous to signed_nat). This makes it impossible to define (for example) a token for creating floating constants.

9 Performance

Both the execution speed and program size were compared between the TDF back-end and the C back-end for several simple benchmarks. The absolute measurements were as follows:

	C back-end		TDF back-end	
program	*code size*	*execution time*	*code size*	*execution time*
library	183600	*(not applicable)*	189824	*(not applicable)*
hello-world	240	*(not applicable)*	176	*(not applicable)*
nfib	512	3.40	448	8.72
bubble-sort	1008	5.32	928	14.61
closures	576	6.50	544	15.27

which give the following relative measurements:

	TDF relative to C	
program	*code size*	*execution time*
library	1.034	*(not applicable)*
hello-world	0.733	*(not applicable)*
nfib	0.875	2.564
bubble-sort	0.921	2.746
closures	0.944	2.349
average		2.553

where: *library* is the runtime support package (written in Dylan); *hello-world* is the usual minimal program; *nfib* generates fibonnacci-like numbers which have the useful property that the result is the number of procedure calls made during the calculation; *bubble-sort* is a simple sorting program applied to a list of 100 integers; and *closures* builds (dynamically) and calls a closure several hundred times.

The "execution time" is the difference in the Unix user-time for the process between the start and end of the benchmark. The "code size" is the sum of the text and data sizes for the ".o" file just prior to the final loading.

Based on the return value from nfib, the speed of the code generated by the two back ends is approximately 939 calls-per-second for the C, and 366 calls-per-second for the TDF, back-end. This difference in performance is due to several simplifications in the TDF back-end which are currently being corrected. The TDF back-end should soon produce code that runs faster than that produced by the C back-end.

10 Conclusions

Our experience with TDF as a delivery vehicle for Dylan programs has so far been encouraging. Although some minor problems have already been encountered, and others are foreseen, these are not likely to seriously affect the Dylan producer. Many of these difficulties have been due to the difficulty of generating "natural" TDF because of restrictions imposed on the producer from parts of the compiler that are shared between many back-ends.

It is important to be aware that the initial producer makes no attempt to implement the more complex language features (integrated garbage collection and threads are two important examples) at the TDF level. However, we do not feel that this is a serious omission since the demands which these will make on TDF have already been predicted thoroughly elsewhere [Man93b] [MM94].

A The Benchmarks

This appendix presents the source code for the benchmarks used to produce the timings given in section 9, and the code generated by both the TDF and C back-ends.

A.1 Dylan Source

The benchmarks were written for a prefix Dylan compiler, and so they are presented using that notation rather than the more recent infix equivalent.

bubble-sort

```
(define-method make-big-list (size)
  (if (= size 0)
      ()
      (pair size (make-big-list (- size 1)))))

(define-method bubble-sort ((list <list>))
  (bind ((tmp #f))
    (until (= tmp (head list))
      (set! tmp (head list))
      (for ((prev = list then (tail prev))
            (next = (tail list) then (tail next))
            (until (empty? next)))
        (when (< (head prev) (head next))
          (set! tmp (head prev))
          (set! (head prev) (head next))
          (set! (head next) tmp))))
    list))

(bubble-sort (make-big-list 100))
```

nfib

```
(define-method nfib (x)
  (if (< x 1)
      1
      (+ 1 (+ (nfib (- x 2))
              (nfib (- x 1))))))

(nfib 15)
```

closures

```
(define-method make-counter ()
  (bind ((x 0))
    (method () (set! x (+ 1 x)))))
```

```
(define-method closure-tests ()
  (for ((n from 0 below 5000))
    ((make-counter))))

(closure-tests)
```

A.2 Generated TDF

The following examples of generated TDF follow the notation understood by TNC (the TDF Notation Compiler). To reduce these examples to a manageable size all declarations of external variables and identifiers, and all declarations pertaining to FASL literals and literal initialisation, have been ignored in the code quoted here.

bubble-sort

```
(make_id_tagdef z_make__big__list__LOC
   (make_proc pz
      pfn - function
      pz - next_methods
      pz - z_size_L_
   | - (sequence
      (variable var_acc TST__0 UNBOUND (sequence
      (variable var_acc z_list_24_M_ UNBOUND (sequence
      (labelled block_intro LBL__0 LBL__1 | (goto block_intro)
            (sequence (make_null_ptr (alignment top)) # block_intro:
      (assign (lval z_list_24_M_) (rval z__pc_empty__list))
      (goto LBL__0))
            (sequence (make_null_ptr (alignment top)) # LBL__0:
      (assign (lval TST__0) (apply_proc pz (lval CALL2) (rval z__pc_equals_qm_)
                                                (rval z_size_L_) (rval z__pc_0)))
      (goto_if_true (rval TST__0) LBL__1)
      (assign (lval z_list_24_M_) (apply_proc pz (lval CALL2) (rval z_pair)
                                                    (rval z_size_L_) (rval z_list_24_M_)))
      (assign (lval z_size_L_) (apply_proc pz (lval CALL2) (rval z_binary__)
                                                  (rval z_size_L_) (rval z__pc_1)))
      (goto LBL__0))
            (sequence (make_null_ptr (alignment top)) # LBL__1:
      (assign (lval z__pc_number__values) (raw_integer 1))
      (return (rval z_list_24_M_))
      )))))))))

(make_id_tagdef z_bubble__sort__LOC
   (make_proc pz
      pfn - function
      pz - next_methods
      pz - z_c_L_
   | - (sequence
```

```
(variable var_acc ARG__5 UNBOUND (sequence
(variable var_acc ARG__4 UNBOUND (sequence
(variable var_acc TST__2 UNBOUND (sequence
(variable var_acc ARG__3 UNBOUND (sequence
(variable var_acc ARG__2 UNBOUND (sequence
(variable var_acc TST__1 UNBOUND (sequence
(variable var_acc ARG__1 UNBOUND (sequence
(variable var_acc TST__0 UNBOUND (sequence
(variable var_acc ARG__0 UNBOUND (sequence
(variable var_acc z_next_30_M_ UNBOUND (sequence
(variable var_acc z_prev_30_M_ UNBOUND (sequence
(variable var_acc z_tmp_27_M_ UNBOUND (sequence
(labelled block_intro LBL__2 LBL__3 LBL__4 LBL__5 LBL__6 LBL__7 | (goto block_intro)
      (sequence (make_null_ptr (alignment top)) # block_intro:
(assign (lval z_tmp_27_M_) (rval z__pc_false))
(goto LBL__2))
      (sequence (make_null_ptr (alignment top)) # LBL__2:
(assign (lval ARG__0) (apply_proc pz (lval CALL1) (rval z__pc_head) (rval z_c_L_)))
(assign (lval TST__0) (apply_proc pz (lval CALL2) (rval z__pc_equals_qm_)
                                                  (rval z_tmp_27_M_) (rval ARG__0)))
(goto_if_true (rval TST__0) LBL__7)
(assign (lval z_tmp_27_M_) (apply_proc pz (lval CALL1) (rval z__pc_head) (rval z_c_L_)))
(assign (lval ARG__1) (rval z_c_L_))
(assign (lval z_next_30_M_) (apply_proc pz (lval CALL1) (rval z__pc_tail) (rval z_c_L_)))
(assign (lval z_prev_30_M_) (rval ARG__1))
(goto LBL__3))
      (sequence (make_null_ptr (alignment top)) # LBL__3:
(assign (lval TST__1) (apply_proc pz (lval CALL1) (rval z__pc_null_qm_) (rval z_next_30_M_)))
(goto_if_true (rval TST__1) LBL__6)
(assign (lval ARG__2) (apply_proc pz (lval CALL1) (rval z__pc_head) (rval z_prev_30_M_)))
(assign (lval ARG__3) (apply_proc pz (lval CALL1) (rval z__pc_head) (rval z_next_30_M_)))
(assign (lval TST__2) (apply_proc pz (lval CALL2) (rval z_binary_lt_)
                                                  (rval ARG__2) (rval ARG__3)))
(goto_if_true (rval TST__2) LBL__5)
(goto LBL__4))
      (sequence (make_null_ptr (alignment top)) # LBL__4:
(assign (lval ARG__4) (apply_proc pz (lval CALL1) (rval z__pc_tail) (rval z_prev_30_M_)))
(assign (lval z_next_30_M_) (apply_proc pz (lval CALL1) (rval z__pc_tail)
                                          (rval z_next_30_M_)))
(assign (lval z_prev_30_M_) (rval ARG__4))
(goto LBL__3))
      (sequence (make_null_ptr (alignment top)) # LBL__5:
(assign (lval z_tmp_27_M_) (apply_proc pz (lval CALL1) (rval z__pc_head)
                                                        (rval z_prev_30_M_)))
(assign (lval ARG__5) (apply_proc pz (lval CALL1) (rval z__pc_head) (rval z_next_30_M_)))
(apply_proc pz (lval CALL2) (rval z__pc_head__setter) (rval ARG__5) (rval z_prev_30_M_))
(apply_proc pz (lval CALL2) (rval z__pc_head__setter)
```

```
                    (rval z_tmp_27_M_) (rval z_next_30_M_))
    (goto LBL__4))
        (sequence (make_null_ptr (alignment top)) # LBL__6:
    (goto LBL__2))
        (sequence (make_null_ptr (alignment top)) # LBL__7:
    (assign (lval z__pc_number__values) (raw_integer 1))
    (return (rval z_c_L_))
    )))))))))))))))))))))))))))))
```

nfib

```
(make_id_tagdef z_nfib__LOC
  (make_proc pz
    pfn - function
    pz - next_methods
    pz - z_x_L_
  | - (sequence
    (variable var_acc RET__0 UNBOUND (sequence
    (variable var_acc ARG__4 UNBOUND (sequence
    (variable var_acc ARG__3 UNBOUND (sequence
    (variable var_acc ARG__2 UNBOUND (sequence
    (variable var_acc ARG__1 UNBOUND (sequence
    (variable var_acc ARG__0 UNBOUND (sequence
    (variable var_acc TST__0 UNBOUND (sequence
    (labelled block_intro LBL__8 LBL__9 | (goto block_intro)
        (sequence (make_null_ptr (alignment top)) # block_intro:
    (assign (lval TST__0) (apply_proc pz (lval CALL2) (rval z_binary_lt_)
                                        (rval z_x_L_) (rval z__pc_1)))
    (goto_if_true (rval TST__0) LBL__9)
    (assign (lval ARG__0) (apply_proc pz (lval CALL2) (rval z_binary__)
                                        (rval z_x_L_) (rval z__pc_2)))
    (assign (lval ARG__2) (apply_proc pz (lval CALL1) (rval z_nfib) (rval ARG__0)))
    (assign (lval ARG__1) (apply_proc pz (lval CALL2) (rval z_binary__)
                                        (rval z_x_L_) (rval z__pc_1)))
    (assign (lval ARG__3) (apply_proc pz (lval CALL1) (rval z_nfib) (rval ARG__1)))
    (assign (lval ARG__4) (apply_proc pz (lval CALL2) (rval z_binary_pl_)
                                        (rval ARG__2) (rval ARG__3)))
    (assign (lval RET__0) (apply_proc pz (lval CALL2) (rval z_binary_pl_)
                                        (rval z__pc_1) (rval ARG__4)))
    (goto LBL__8))
        (sequence (make_null_ptr (alignment top)) # LBL__8:
    (return (rval RET__0))
        (sequence (make_null_ptr (alignment top)) # LBL__9:
    (assign (lval RET__0) (rval z__pc_1))
    (assign (lval z__pc_number__values) (raw_integer 1))
    (goto LBL__8)
    )))))))))))))))))))
```

closures

```
(make_id_tagdef z_closure__tests__LOC
   (make_proc pz
      pfn - function
      pz - next_methods
   | - (sequence
      (variable var_acc ARG__1 UNBOUND (sequence
      (variable var_acc ARG__0 UNBOUND (sequence
      (variable var_acc FUN__0 UNBOUND (sequence
      (variable var_acc TST__0 UNBOUND (sequence
      (variable var_acc z_n__increment_L_ UNBOUND (sequence
      (variable var_acc z_n__limit_L_ UNBOUND (sequence
      (variable var_acc z_n_38_M_ UNBOUND (sequence
      (labelled block_intro LBL__10 LBL__11 | (goto block_intro)
            (sequence (make_null_ptr (alignment top)) # block_intro:
      (assign (lval z_n_38_M_) (rval z__pc_0))
      (assign (lval z_n__limit_L_) (rval z_LIT__5))
      (assign (lval z_n__increment_L_) (rval z__pc_1))
      (goto LBL__10))
            (sequence (make_null_ptr (alignment top)) # LBL__10:
      (assign (lval TST__0) (apply_proc pz (lval CALL2) (rval z_binary_lt_)
                                                (rval z_n_38_M_) (rval z_n__limit_L_)))
      (goto_if_false (rval TST__0) LBL__11)
      (assign (lval FUN__0) (apply_proc pz (lval z_make__counter__LOC)
                                                UNBOUND (rval z__pc_false)))
      (apply_proc pz (lval CALL0) (rval FUN__0))
      (assign (lval ARG__0) (apply_proc pz (lval CALL2) (rval z_binary_pl_)
                                                (rval z_n_38_M_) (rval z_n__increment_L_)))
      (assign (lval ARG__1) (rval z_n__limit_L_))
      (assign (lval z_n__increment_L_) (rval z_n__increment_L_))
      (assign (lval z_n_38_M_) (rval ARG__0))
      (assign (lval z_n__limit_L_) (rval ARG__1))
      (goto LBL__10))
            (sequence (make_null_ptr (alignment top)) # LBL__11:
      (assign (lval z__pc_number__values) (raw_integer 1))
      (return (rval z__pc_false))
      )))))))))))))))))))

(local make_id_tagdef z_method__LOC__0_36_M_
   (make_proc pz
      pfn - function
      pz - next_methods
   | - (sequence
      (variable var_acc environment (get_e (rval function)) (sequence
      (labelled block_intro | (goto block_intro)
            (sequence (make_null_ptr (alignment top)) # block_intro:
```

```
        (assign (lref (primitive_element environment 0)) (rval z__pc_42))
        (assign (lval z__pc_number__values) (raw_integer 1))
        (return (rref (primitive_element environment 0)))
        )))))))

(make_id_tagdef z_make__counter__LOC
    (make_proc pz
        pfn - function
        pz - next_methods
    | - (sequence
        (variable var_acc RET__0 UNBOUND (sequence
        (variable var_acc ARG__1 UNBOUND (sequence
        (variable var_acc ARG__0 UNBOUND (sequence
        (variable var_acc z_x_34_M_ (make_box UNBOUND) (sequence
        (labelled block_intro | (goto block_intro)
                (sequence (make_null_ptr (alignment top)) # block_intro:
        (assign (lref (rval z_x_34_M_)) (rval z__pc_0))
        (assign (lval ARG__0)
                    (apply_proc pz (lval z_primitive__make__xep__closure)
                        (lval z_method__LOC__0_36_M_)
                        (apply_proc pz (lval z_primitive__make__environment)
                                        (raw_integer 1) (rval z_x_34_M_))
                        (raw_integer 1)))
        (assign (lval ARG__1) (apply_proc pz (lval CALL0) (rval z__pc_list)))
        (assign (lval RET__0)
                    (apply_proc pz (lval CALL)
                        (rval z__pc_make__method) (raw_integer 12) XEP (rval ARG__0)
                        (rval z__pc_k__debug__name) (rval z__pc_false)
                        (rval z__pc_k__number__required) (rval z__pc_0) (rval z__pc_k__rest_qm_)
                        (rval z__pc_false) (rval z__pc_k__key_qm_) (rval z__pc_false)
                        (rval z__pc_k__specializers) (rval ARG__1)))
        (return (rval RET__0))
        )))))))))))))
```

A.3 Generated C

As with the TDF above, the following examples of generated C omit many declarations of external variables for sake of brevity. They are intended merely to show the correspondence between the generated TDF and equivalent C.

bubble-sort

```
Z* z_make__big__list__LOC(Z* function, Z* next_methods, Z* z_size_L_)
    Z *TST__0, *z_list_24_M_;
    z_list_24_M_ = z__pc_empty__list;
  LBL__0:
    TST__0 = CALL2(z__pc_equals_qm_,z_size_L_,z__pc_0);
```

```
    if (TST__0 != z__pc_false)
       goto LBL__1;
    z_list_24_M_ = CALL2(z_pair,z_size_L_,z_list_24_M_);
    z_size_L_ = CALL2(z_binary__,z_size_L_,z__pc_1);
    goto LBL__0;
  LBL__1:
    z__pc_number__values = (Z*)1;
    return(z_list_24_M_);


Z* z_bubble__sort__LOC(Z* function, Z* next_methods, Z* z_c_L_)
    Z *ARG__5, *ARG__4, *TST__2, *ARG__3, *ARG__2, *TST__1, *ARG__1,
      *TST__0, *ARG__0, *z_next_30_M_, *z_prev_30_M_, *z_tmp_27_M_;
    z_tmp_27_M_ = z__pc_false;
  LBL__2:
    ARG__0 = CALL1(z__pc_head,z_c_L_);
    TST__0 = CALL2(z__pc_equals_qm_,z_tmp_27_M_,ARG__0);
    if (TST__0 != z__pc_false)
       goto LBL__7;
    z_tmp_27_M_ = CALL1(z__pc_head,z_c_L_);
    ARG__1 = z_c_L_;
    z_next_30_M_ = CALL1(z__pc_tail,z_c_L_);
    z_prev_30_M_ = ARG__1;
  LBL__3:
    TST__1 = CALL1(z__pc_null_qm_,z_next_30_M_);
    if (TST__1 != z__pc_false)
       goto LBL__6;
    ARG__2 = CALL1(z__pc_head,z_prev_30_M_);
    ARG__3 = CALL1(z__pc_head,z_next_30_M_);
    TST__2 = CALL2(z_binary_lt_,ARG__2,ARG__3);
    if (TST__2 != z__pc_false)
       goto LBL__5;
  LBL__4:
    ARG__4 = CALL1(z__pc_tail,z_prev_30_M_);
    z_next_30_M_ = CALL1(z__pc_tail,z_next_30_M_);
    z_prev_30_M_ = ARG__4;
    goto LBL__3;
  LBL__5:
    z_tmp_27_M_ = CALL1(z__pc_head,z_prev_30_M_);
    ARG__5 = CALL1(z__pc_head,z_next_30_M_);
    CALL2(z__pc_head__setter,ARG__5,z_prev_30_M_);
    CALL2(z__pc_head__setter,z_tmp_27_M_,z_next_30_M_);
    goto LBL__4;
  LBL__6:
    goto LBL__2;
  LBL__7:
    z__pc_number__values = (Z*)1;
```

```
    return(z_c_L_);
```

nfib

```
Z* z_nfib__LOC(Z* function, Z* next_methods, Z* z_x_L_)
    Z *RET__0, *ARG__4, *ARG__3, *ARG__2, *ARG__1, *ARG__0, *TST__0;
    TST__0 = CALL2(z_binary_lt_,z_x_L_,z__pc_1);
    if (TST__0 != z__pc_false)
        goto LBL__9;
    ARG__0 = CALL2(z_binary__,z_x_L_,z__pc_2);
    ARG__2 = CALL1(z_nfib,ARG__0);
    ARG__1 = CALL2(z_binary__,z_x_L_,z__pc_1);
    ARG__3 = CALL1(z_nfib,ARG__1);
    ARG__4 = CALL2(z_binary_pl_,ARG__2,ARG__3);
    RET__0 = CALL2(z_binary_pl_,z__pc_1,ARG__4);
  LBL__8:
    return(RET__0);
  LBL__9:
    RET__0 = z__pc_1;
    z__pc_number__values = (Z*)1;
    goto LBL__8;
```

closures

```
Z* z_closure__tests__LOC(Z* function, Z* next_methods)
    Z *ARG__1, *ARG__0, *FUN__0, *TST__0, *z_n__increment_L_,
      *z_n__limit_L_, *z_n_38_M_;
    z_n_38_M_ = z__pc_0;
    z_n__limit_L_ = z_LIT__5;
    z_n__increment_L_ = z__pc_1;
  LBL__10:
    TST__0 = CALL2(z_binary_lt_,z_n_38_M_,z_n__limit_L_);
    if (TST__0 == z__pc_false)
        goto LBL__11;
    FUN__0 = z_make__counter__LOC(UNBOUND,z__pc_false);
    CALL0(FUN__0);
    ARG__0 = CALL2(z_binary_pl_,z_n_38_M_,z_n__increment_L_);
    ARG__1 = z_n__limit_L_;
    z_n__increment_L_ = z_n__increment_L_;
    z_n_38_M_ = ARG__0;
    z_n__limit_L_ = ARG__1;
    goto LBL__10;
  LBL__11:
    z__pc_number__values = (Z*)1;
    return(z__pc_false);
```

```
static Z* z_method__LOC__0_36_M_(Z* function, Z* next_methods)
    Z** environment = ((FN*)function)->e;
    *((Z**)environment[0]) = (int)(*((Z**)environment[0]))+(int)(Z*)1;
    z__pc_number__values = (Z*)1;
    return(*((Z**)environment[0]));


Z* z_make__counter__LOC(Z* function, Z* next_methods)
    Z *RET__0, *ARG__1, *ARG__0;
    Z** z_x_34_M_ = z_primitive__make__box(UNBOUND);
    *((Z**)z_x_34_M_) = z__pc_0;
    ARG__0 = (Z*)z_primitive__make__xep__closure(
                    z_method__LOC__0_36_M_,
                    z_primitive__make__environment(1,z_x_34_M_),1);
    ARG__1 = CALL0(z__pc_list);
    RET__0 = CALL(z__pc_make__method,12,XEP,ARG__0,z__pc_k__debug__name,
                z__pc_false,z__pc_k__number__required,z__pc_0,
                z__pc_k__rest_qm_,z__pc_false,z__pc_k__key_qm_,
                z__pc_false,z__pc_k__specializers,ARG__1);
    return(RET__0);
```

References

[App92] Apple Computer, “Dylan: an object-oriented dynamic language”, April 1992.

[FC92] M. Foster and I. Currie, “TDF Specification”, DRA Malvern, September 1992.

[Man93a] T. Mann, “TDF Support Required by Lisp”, OMI/GLUE Deliverable 4.2.1, Harlequin Ltd., 1993.

[Man93b] T. Mann, “Initial Evaluation of TDF Support for Garbage Collection”, OMI/GLUE Deliverable 4.2.2a, Harlequin Ltd., 1993.

[MM94] T. Mann and E. Miranda, “Requirements specification for parallel extensions to TDF to support Dylan — including runtime support”, OMI/GLUE Deliverable 5.7.1, Harlequin Ltd., March 1994.