ESPRIT PROJECT 6062

Study of Performance of TDF-Based Threaded Systems

Written by: Tony Mann, Rod Moyse,
Simon Green, Robert Mathews
Harlequin Ltd. ______________________

Issued by: Peter Edwards ______________________
DRA

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: TR5.7.2-01

Document code: Harlequin GLUE5.7.2 - 01

Date of first issue: 1995-09-01
Date of last issue: 1995-09-01

Availability: Confidential

omi/glue

CHANGE HISTORY

This is the first version.

1. Purpose

It is convenient to use threads for many application domains, from servers to GUIs. This is being recognized by modern operating systems, many of which now provide interfaces to threads facilities by means of an API. Some high level languages provide direct support for threads, too.

Dylan [Apple94] is a high level object oriented programming language, with dynamic capabilities, and automatic memory management. It is defined as a small core language with no support for parallelism, and encourages language extensions via libraries. We present here a design for such a library to add threads support to Dylan — the *Simple Dylan Threads Library.*

This document describes a complete design for the threads library for Dylan. It shows how a Dylan program using threads may be compiled to a portable ANDF representation, and how a combination of install-time token libraries and runtime system libraries may be used to implement the system on a particular platform, in terms of threads APIs.

This work is presented as a proof of concept, to show that ANDF is an appropriate medium for implementing an advanced dynamic language with support for threads. Some empirical analysis is made of the impact of ANDF on the performance of programs described in this way.

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

2. Executive Summary

A major goal of Dylan is to support the creation of reusable software components. It is a requirement of Harlequin's implementation of Dylan that these components should be inter-operable with components written in other languages. This has implications for the design of a threads library, since threads of control need not be limited to component boundaries. It is primarily for this reason that the Simple Dylan Threads Library is designed to be implementable in terms of typical threads APIs provided by mainstream operating systems (without constraining the design to any particular platform).

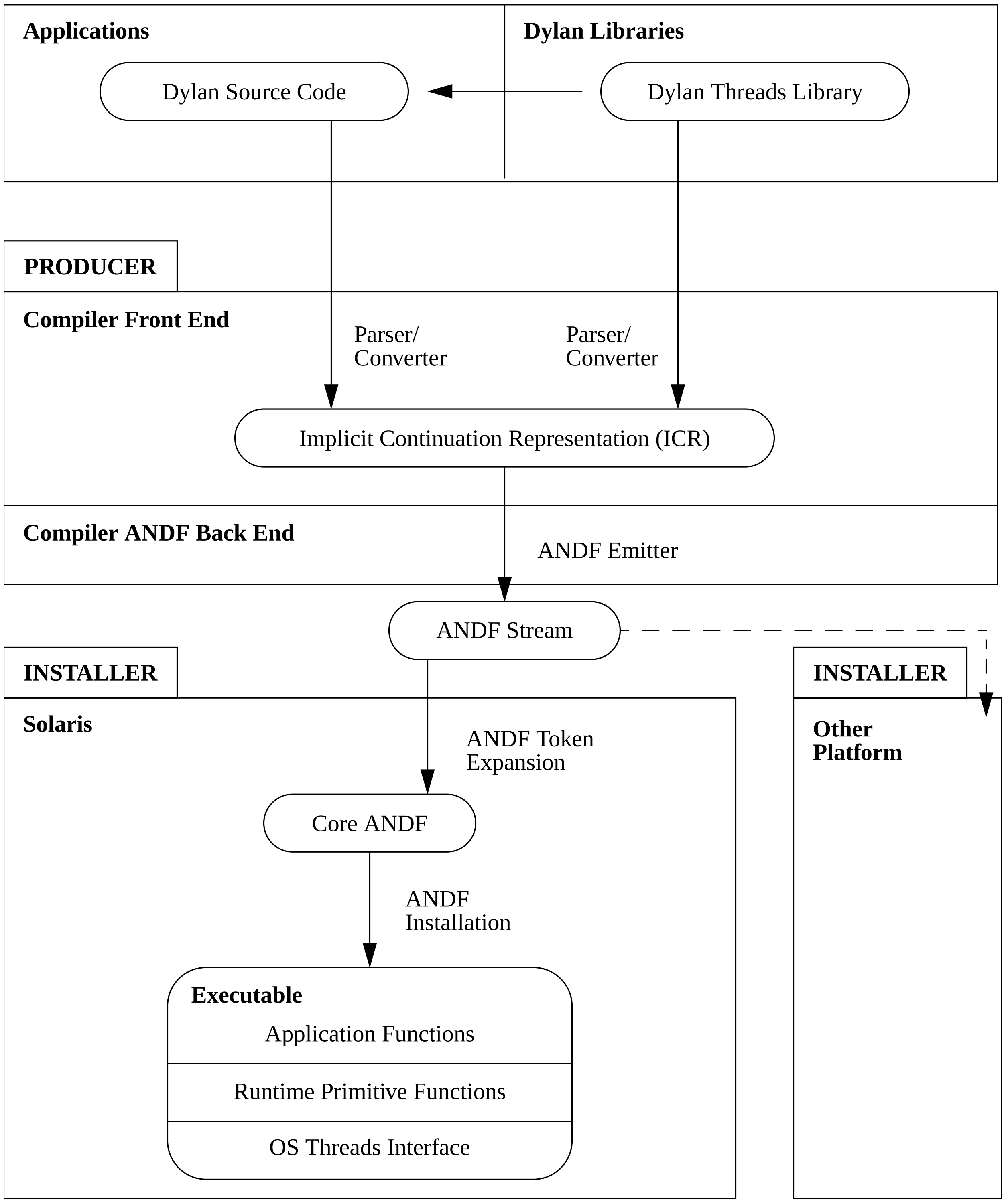
Although the Simple Dylan Threads Library has not been endorsed as a standard, it is hoped that it might evolve into a standard. In keeping with the language of Dylan, it is described as an object oriented class library, although it is not defined at a particularly high level. It is expected that higher level libraries (with support for monitors, for instance) can be built in a layered fashion from the facilities of the simple library. The library is designed to be portable — and a mechanism for achieving portability with ANDF is described.

ANDF [TDF92] has no direct support for parallelism, but earlier studies ([Edwards93], [ME94]) have shown how a combination of core ANDF and ANDF tokens may be used to support parallel features, provided that appropriate expanders for the tokens are available on the installation platform. These token expansions might describe calls to threads APIs, or might be specially recognized by the installer.

The set of ANDF tokens used to manage portability for the Dylan threads library is carefully crafted to describe the features of the library. It is designed to be ultimately mapped onto operating system APIs for threads, but is not designed to be similar to any particular API. Conceptually, it describes an API for threads support from the Dylan runtime system, and it might be a reasonable choice on some platforms to implement it in that way. The library has been tested on Solaris using such a runtime system to provide the threads support. The Solaris implementation is described in this document.

The effect of ANDF on the general implementation of Dylan with multiple threads is discussed in [ME94]. Since ANDF does not provide the direct support for threads or synchronization, it has little effect on the performance of a threads system built using it. ANDF is a convenient layer of abstraction over the differences between platforms, and plays little part in the implementation choices for any platform. There are features in the Dylan threads library, however, which can only be implemented efficiently on some systems if special hardware instructions are used. ANDF may impose a cost for these features on some target platforms if there is no appropriate token expansion which will cause the installer on the target to generate these hardware instructions.

Figure 1. Processing of Dylan Threads Constructs



3. Simple Dylan Threads Library

3.1 Introduction

The goals of the simple Dylan threads library, in order of importance, are:

- Maps easily and efficiently onto OS provided threads facilities for common OSs
- Permits all common thread models, including co-operative, pre-emptive and concurrent
- Makes use of the language features of Dylan
- Provides enough functionality to implement more advanced features portably in terms of the simple features provided here

The library is called **threads**. All names are exported from the module called **threads**.

3.2 Multi-thread Semantics

This library provides multiple threads of control within a single space of objects and module variables. Each thread runs in its own independent stack. The mechanism by which the threads are scheduled is not specified, and it is not possible to determine how the execution of instructions by different threads will be interleaved. No mechanism is provided to call a function on a thread other than the current thread. Neither is there a mechanism to signal an exception on a thread other than the current thread.

3.2.1 Implicit Synchronization

The threads library guarantees that assignments to slots and to variables are atomic. The time at which that assignment becomes visible to another thread is undefined, unless the two threads synchronize explicitly. I.e. after an assignment, but before explicit synchronization, another thread will either see the old value or the new value of the location. There is no possibility of seeing a half-way state.

The ordering of visibility of side-effects performed in other threads is undefined, unless explicit synchronization is used. Implementations of the library may guarantee that the visibility of side-effects performed by another thread is ordered according to the control flow of the other thread (*strong ordering*), but multi-processor implementations may not be strongly ordered. Portable code should not assume strong ordering, and should use explicit synchronization where order of side effects is important. There is currently no library introspection facility to determine if the implementation is strong or weak ordered.

Because of the possibility of weak ordering, the compiler is free to assume that the effects of other threads may be ignored between explicit synchronization points — and may perform any optimizations which preserve the semantics of a single-thread model regardless of their effects on other threads (e.g. common sub-expression elimination, or changing the order of evaluation).

3.2.2 Explicit Synchronization

The threads library provides a variety of synchronization facilities, described below. These facilities include mutual-exclusion locks, semaphores and notifications. Each facility guarantees that when synchronization has been achieved, then all the side effects of another thread are visible, at least up to the point where that other thread last released the synchronization facility.

An appropriate synchronization must be used to guard side-effects on state if there is any possibility of those side-effects either being corrupted by another thread or corrupting another thread. For example, a function which assigns to two slots of an object may require the use of a lock to guarantee that other threads never observe the object in a partly updated state.

3.2.3 Conditional Update

In addition to the synchronization primitives, the library provides a conditional update mechanism which is not synchronized, but which tests whether the value in a variable or slot has changed and atomically updates it if not.

By using conditional updates, a thread can confirm (or deny) that there has been no interference from other threads, without any need for a blocking operation. This is more efficient for those circumstances where interference is not disastrous and it is possible to re-compute the update.

For example, a function which increments the value of a variable might use a conditional update to store the new value into place, in order to guarantee a numeric sequence for the variable. In this example, the function might loop until the conditional update has succeeded.

It is possible to achieve synchronization by looping until a conditional update is successful. This is not recommended, because the *busy-waiting* state during the loop may disallow other threads from running. Normally, conditional update should be used only when it is expected to succeed. If it is likely that the conditional update might fail multiple times around the loop, then either the number of times around the loop should be limited, or a blocking function from the **threads** library should be used within the loop.

3.2.4 Dynamic Environment

Dylan has an implicit notion of a dynamic environment, corresponding to language constructs with *dynamic extent*. For example, the **block** construct can introduce *cleanup-clauses*, and the *body* of the block is executed in a dynamic environment in which those cleanup-clauses are active. *Handlers* and *exit procedures* are other examples of language features related to the dynamic environment. The threads library introduces a new construct which affects the dynamic environment called **fluid-bind** (defined below).

The dynamic environment is defined to be thread-local. When a new thread is created, it starts with a fresh dynamic environment. It is an error to attempt to use a handler or a non-local exit function belonging to another thread. It is impossible to use an unwind-protect cleanup from another thread.

Although the binding of condition handlers only affects the dynamic environment of the current thread, unhandled conditions are passed to the global generic function **default-handler**. Potentially this function might *call the debugger*. The threads library does not define what this term might mean.

Note that in Dylan, unlike in C and C++, *lexical* variables (i.e. local, or LET bound variables) have indefinite extent, and are not bound in the dynamic environment. In general, the variables are potentially global — and programmers may need to explicitly synchronize accesses to them accordingly. The introduction of threads to Dylan does not affect the efficiency of lexical variables which do not require explicit synchronization.

3.2.5 Fluid Binding

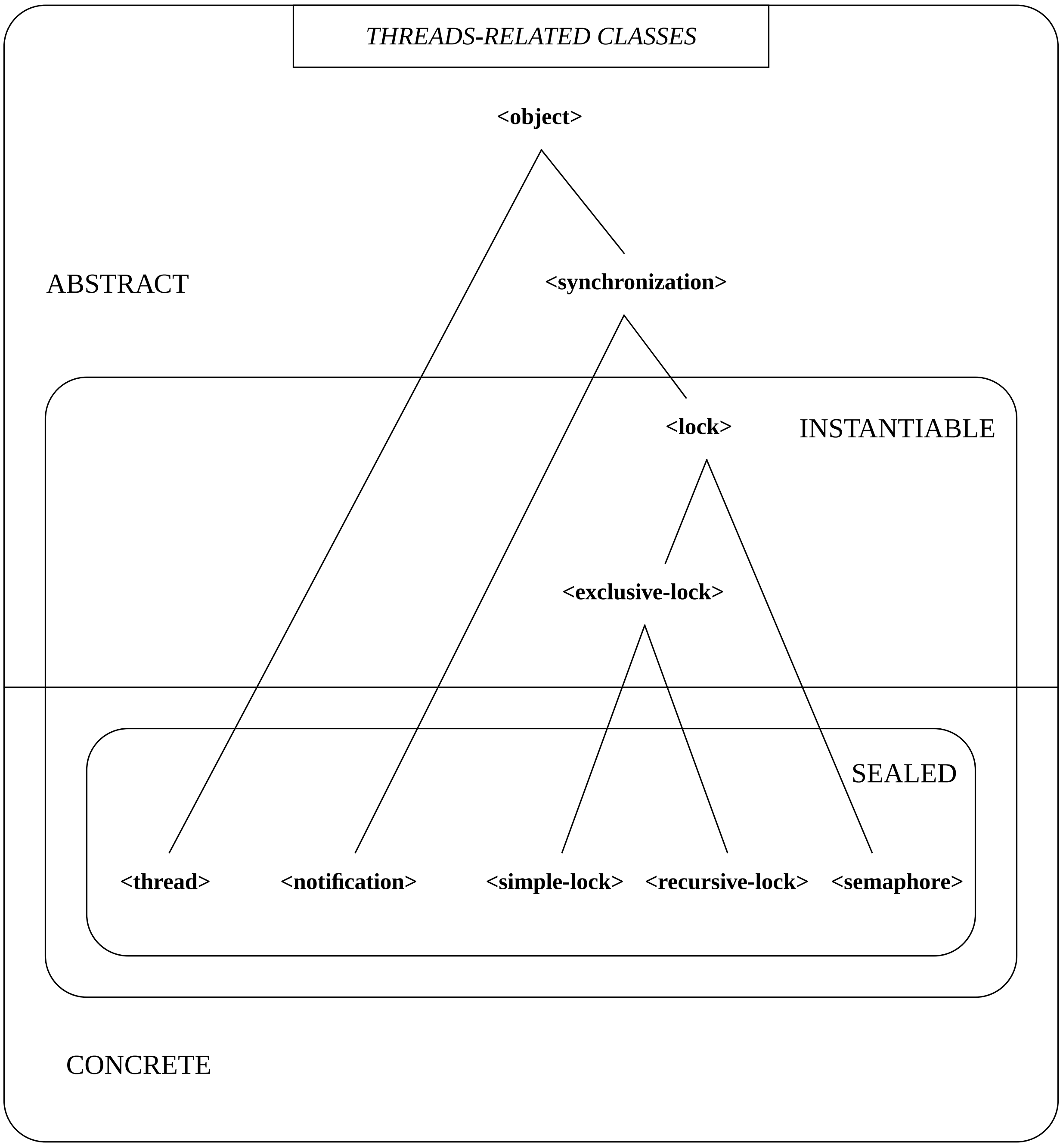
The threads library provides a new defining form, **fluid-variable**, for defining module variables which are suitable for *fluid binding*. A *binding* is a mapping between a variable and a *value-cell* which holds the variable's value. A *fluid* binding is a binding which has dynamic extent, and shadows any outermost bindings. Fluid variables may have their bindings changed with **fluid-bind**, which is a primitive mechanism for creating new bindings with dynamic extent. Fluid variables are visible in the module namespace, but their bindings (and hence value-cells) exist in the dynamic environment. The mechanism includes a small amount of extensibility, and it is hoped that more complex mechanisms (such as fluid-binding of slots) can be built from it.

Fluid variables contain a private value-cell per thread. Whenever a thread is created, the value of the fluid variable is implicitly bound in the dynamic environment of that thread, and its initial value is the thread-independent value resulting from evaluation the initialization expression of the fluid variable definition. The fact that the value-cells are always thread-local means that fluid variables may be used as thread local variables, even if they are not explicitly re-bound. The thread locality of fluid variables is an important feature for programs which make use of threads, and it is for this reason that fluid variables are defined in the threads library.

The thread-local nature of the initial binding of fluid variables may not be optimal for all problem domains. For instance a shared, global, outermost binding may be desirable, or alternatively, a thread may want to inherit current bindings from the parent thread at creation time, giving a “fork” type model of state inheritance. These alternatives are not implemented in this library. In principle, it should be possible to implement either or both of these mechanisms at a later date, possibly using adjectives to the **fluid-variable** definition form to specify the mechanism required, Maybe it would be useful to introduce the definition form **fluid-constant** one day, too.

3.3 Threads-Related Class Hierarchy

Figure 2. Dylan Threads-Related Classes



3.4 Library Reference

3.4.1 <thread>

<thread> [Instantiable Sealed Class]

Superclasses

<object>

Required-init-keywords

function:	A **<function>** which will be called with no arguments in the empty dynamic environment of the new thread. This function will be runnable immediately. If desired, the new thread may be suspended (almost) immediately on creation by arranging for it to synchronize on an unavailable resource on entry to the function.

Init-keywords

priority:	The scheduling priority of the thread, which is a signed integer. The higher the value, the greater the priority. **0** is the default value, and is the value of **$normal-priority**, one of several constants which correspond to useful priority levels. The library currently offers no way to change the priority of a thread dynamically.
name:	If supplied, this should be a **<string>**, which names the thread object for purposes which might include debugging, or supporting inter-process communication in a future version of the library.

Description

The class representing a thread of control. Note that there is currently no ability to introspect or set the running state or priority of a thread. These abilities are considered to be related to development environments, debuggers or schedulers and were deliberately omitted. There is also no way to terminate a thread, mainly because there is no clear implementation technique which is both portable to common operating systems, and will also ensure that cleanups happen. There is currently no way to signal a condition on another thread. If there was, it might prove a safe way to terminate threads.

Operations

The class **<thread>** provides the following operations:

thread-name	Returns the name of a thread, or **#f** if no name was supplied.
join-thread	Blocks until one of the specified threads has terminated, and returns the values of its function.

Associated Constants

The following constants (in order of increasing value) may be useful as priority values for threads:

$low-priority

$background-priority

$normal-priority

$interactive-priority

$high-priority

thread-name [Function]

Signature

(thread :: <thread>) => (name :: false-or(<string>))

Description

Returns the name of the thread as a **<string>**, or, if the thread is unnamed, returns **#f**.

join-thread [Function]

Signature

(thread1 :: <thread>, #rest more-threads) => (thread, #rest results)

Arguments

thread1	A thread object to wait for.
more-threads	More thread objects to wait for.

Values

thread	The thread that exited
results	The values returned from the thread that was joined

Description

Returns one of the supplied threads which has finished (because its function has returned), along with any values returned by the thread. **join-thread** will block if necessary, while waiting for a thread to finish. It is an error to pass a thread to **join-thread** if it has already been joined in a previous call to **join-thread**. It is an error to pass a thread to **join-thread** if that thread is also being processed by another simultaneous call to **join-thread** from another thread.

Exceptions

join-thread may signal the following condition:

<duplicate-join-error>
A condition of this class (a subclass of **<error>**) may be signalled when a thread is passed to **join-thread**, if that thread has been previously joined by an earlier

call to **join-thread**, or if that thread is currently active in another call to **join-thread**.

thread-yield [Function]

Signature

() => ()

Description

Causes the current thread to yield control to the thread scheduler. This may have the effect of allowing other threads to run — and may be essential to avoid deadlock in a co-operative scheduling environment.

current-thread [Function]

Signature

() => (thread :: <thread>)

Description

Returns the current thread.

3.4.2 <synchronization>

<synchronization> [Abstract Open Class]

Superclasses

<object>

Init-keywords

name: If supplied, this should be a **<string>**, which names the synchronization object for purposes which might include debugging, or supporting inter-process communication in a future version of the library.

Operations

The class **<synchronization>** provides the following operations:

wait-for Block until synchronization can be achieved.

release Release the object to make it available for synchronization.

synchronization-name
Returns the name of the synchronization object.

Description

The class of objects which are used for inter-thread synchronization. There is currently no explicit mechanism in the library to block on a number of synchronization objects simultaneously, until synchronization can be achieved with one of them. This mechanism may be achieved instead by creating a new thread to wait for each synchronization object, and arranging for each thread to release a notification once synchronization has been achieved.

wait-for [Open Generic Function]

Signature

(object :: <synchronization>, #key timeout = #f) => (success)

Arguments

object	Synchronization object to wait for.
timeout	Time-out interval. If the value is **#f** (the default), the time-out interval never elapses. Otherwise the value should be a **<real>**, corresponding to the desired interval in seconds.

Values

success	A true value if synchronization was achieved, or **#f** to indicate that a time-out occurred.

Description

This is the basic blocking primitive of the threads library. The function blocks until the object is available and synchronization can be achieved, or the time-out interval has expired. A non-blocking synchronization may be attempted by specifying a timeout of 0. Individual methods may adjust the state of the synchronization object on synchronization. The function returns a true value if synchronization is achieved before this interval elapses; otherwise, it returns **#f.** Once synchronization has been achieved, the side-effects of other threads will be visible, at least up to the point where they last passed the object to **release**.

release [Open Generic Function]

Signature

(object :: <synchronization>, #key) => ()

Arguments

object	Synchronization object to release.

Values

None

Description

Releases the supplied synchronization object, potentially making it available to other threads. Individual methods describe what this means for each class of synchronization. This function does not block for any of the subclasses of **<synchronization>** provided by the library.

Exceptions

Individual methods may provide their own policies.

synchronization-name [Open Generic Function]

Signature

(object :: <synchronization>) => (name :: false-or(<string>))

Description

Returns the name of the synchronization object, if it was specified with the **name:** initialization keyword. Otherwise **#f** is returned.

3.4.3 <lock>

<lock> [Abstract Instantiable Class]

Superclasses

<synchronization>

Description

Locks are synchronization objects which change state on synchronization (using **wait-for**), and revert state on **release**. It is normally necessary for programs to ensure that locks are released, otherwise there is the possibility of deadlock. Locks may be used to restrict the access of other threads to shared resources between the synchronization and the release. It is common for a protected operation to be performed by body of code which is evaluated in a single thread between synchronization and release. A macro **with-lock** is provided for this purpose. When a thread uses a lock for mutual-exclusion in this way, the thread is said to own the lock.

<lock> has no direct instances; calling make on **<lock>** returns an instance of **<simple-lock>**.

Operations

The class **<lock>** provides the following operations:

with-lock	Execute a body of code between **wait-for** and **release** operations.

with-lock [Macro]

Syntax

```
with-lock (lock :: <lock>, #key keys)
    body
[failure failure-expr]
end
```

Exceptions

with-lock may signal a condition of the following class (a subclass of <**serious-condition**>):

<timeout-expired>
This is signalled when **with-lock** did not succeed in claiming the lock within the timeout period.

Description

Execute the body with the lock held. If a **failure** clause is supplied, then it will be evaluated and its values returned from **with-lock** if the lock cannot be claimed (because a timeout occurred). The default, if no **failure** clause is supplied, is to signal an exception of class <**timeout-expired**>. If there is no failure, **with-lock** returns the results of evaluating the body. As an example, if no **failure** clause is supplied, the macro expands into code equivalent to the following:

```
if (   wait-for(lock, keys ...))
       block ()
           body ...
       cleanup
           release(lock)
       end block
else
       signal(make(<timeout-expired>, synchronization: lock)
end if
```

<semaphore> [Instantiable Sealed Class]

Superclasses

<lock>

Description

<semaphore> is a class representing a traditional counting semaphore. An instance of <**semaphore**> contains a counter in its internal state. Calling **release** on a semaphore will increment the internal count. Calling **wait-for** on a semaphore will decrement the internal count, unless it is zero, in which case the thread will block until another thread releases the semaphore. Semaphores are less efficient than exclusive locks, but they have asynchronous properties which may be useful (e.g. for managing queues, or pools of shared resources). Semaphores may be released by any thread, so there is no built-in concept of a thread owning a semaphore. It is not necessary for a thread to release a semaphore after waiting for it — although semaphores may be used as locks if they do.

Init-keywords

initial-count: This should be a non-negative integer, corresponding to the initial state of the internal counter. The default value is 0.

maximum-count:
This should be a non-negative integer corresponding to the maximum permitted value of the internal counter. The default value is the largest value supported by the implementation, which is the value of the constant **$semaphore-maximum-count-limit**. This constant will not be smaller that **10000**.

release [Sealed Method]

Signature

(object :: <semaphore>, #key) => ()

Exceptions

release may signal a condition of the following class (which is a subclass of **<error>**).

<count-exceeded-error>
This may signalled when an attempt is made to release a **<semaphore>** when the internal counter is already at its maximum count.

<exclusive-lock> [Abstract Instantiable Class]

Superclasses

<lock>

Description

<exclusive-lock> is the class of locks for which threads cannot change the state of the lock unless they own it. An instance of **<exclusive-lock>** may only be locked by one thread at a time (by calling **wait-for** on the lock). Once locked, any attempt by any other thread to wait for the lock will cause that thread to block. It is an error for a thread to release an **<exclusive-lock>** if it does not own the lock. The notion of ownership is directly supported by the class, and a thread may test whether an **<exclusive-lock>** is currently owned.

<exclusive-lock> has no direct instances; calling make on **<exclusive-lock>** returns an instance of **<simple-lock>**.

Init-keywords

Operations

The class **<exclusive-lock>** provides the following operations:

owned? Tests to see if the lock is owned by the current thread.

release [GF Method]

Signature

(object :: <exclusive-lock>, #key) => ()

Exceptions

release may signal a condition of the following class (which is a subclass of **<error>**).

<not-owned-error>
This may be signalled when an attempt is made to release an **<exclusive-lock>** when the lock is not owned by the current thread.

owned? [Generic Function]

Signature

(object :: <simple-lock>) => (owned? :: <boolean>)

Description

A predicate to test whether the simple lock is locked by the current thread.

<recursive-lock> [Instantiable Sealed Class]

Superclasses

<exclusive-lock>

Description

A thread may lock a **<recursive-lock>** multiple times, recursively, but the lock must later be released the same number of times. The lock will be freed on the last of these releases.

<simple-lock> [Instantiable Sealed Class]

Superclasses

<exclusive-lock>

Description

<simple-lock> is the class representing the most simple and efficient mutual exclusion synchronization primitive. It is an error to recursively lock a **<simple-lock>**. An attempt to do so might result in an error being signalled, or deadlock occurring.

3.4.4 <notification>

<notification> [Instantiable Sealed Class]

Superclasses

<synchronization>

Description

Instances of **<notification>** are synchronization objects which may be used to notify threads of a change of state elsewhere in the program. Notifications are used in association with locks, and are sometimes called *condition variables*. They may be used to support the sharing of data between threads using *monitors*. Each **<notification>** is permanently associated with a **<simple-lock>**, although the same lock may be associated with many notifications. Threads wait for the change of state to be notified by calling **wait-for**. Threads notify other threads of the change of state by calling **release**.

Required-init-keywords

lock:	This must be an instance of **<simple-lock>**. The lock is then associated with the notification, and it is only possible to wait for or release the notification if the lock is owned.

Operations

The class **<notification>** provides the following operations:

associated-lock	Returns the lock associated with the notification object.
wait-for	Wait for the notification of the change in state. The associated lock must be owned, and is atomically released before synchronization, and reclaimed after.
release	Notify the change of state to a single waiting thread. This has no effect on the associated lock, which must be owned.
release-all	Notify the change of state to all waiting threads. This has no effect on the associated lock, which must be owned.

associated-lock [Function]

Signature

(notification :: <notification>) => (lock :: <simple-lock>)

Arguments

notification	A notification object.

Values

lock	The lock associated with the notification.

wait-for [Sealed Method]

Signature

(notification :: <notification>, #key timeout = #f) => (success)

Description

Wait for the change of state indicated by the notification. The associated lock must be owned, and is atomically released before the wait, and atomically claimed again after the wait. Note that the state should be tested again once **wait-for** has returned, because there may have been a delay between the **release** of the notification and the claiming of the lock, and the state may have been changed during that time. If a timeout is supplied, then this will be used for waiting for the release of the notification only. **wait-for** will always wait for the lock with no timeout, and it is guaranteed that the lock will be owned on return. **wait-for** will return **#f** if the notification wait times out.

Exceptions

wait-for may signal a condition of the following class (which is a subclass of **<error>**).

<not-owned-error>
This may be signalled when an attempt is made to wait for a notification when the associated lock is not owned by the current thread.

release [Sealed Method]

Signature

(notification :: <notification>, #key) => ()

Arguments

notification	Notification to be released.

Description

Releases the notification, announcing the change of state to one of the threads which are blocked and waiting for it. The choice of which thread receives the notification is undefined. The receiving thread may not be unblocked immediately, because it must first claim ownership of the associated lock.

Exceptions

Release may signal a condition of the following class (which is a subclass of **<error>**).

<not-owned-error>
This may be signalled when an attempt is made to release a notification when the associated lock is not owned by the current thread.

release-all [Function]

Signature

(notification :: <notification>, #key) => ()

Arguments

notification	Notification to be released.

Description

Releases the notification, announcing the change of state to all threads which are blocked and waiting for it. Those threads will then necessarily have to compete for the associated lock.

Exceptions

release-all may signal a condition of the following class (which is a subclass of **<error>**).

<not-owned-error>
This may be signalled when an attempt is made to release a notification when the associated lock is not owned by the current thread.

3.4.5 Timers

sleep [Function]

Signature

(interval :: <real>) => ()

Arguments

interval	Sleep time in seconds.

Description

sleep causes the current-thread to block for the specified number of seconds.

3.4.6 Conditional Update

atomic-variable [Definition]

Syntax

define atomic-variable *bindings* = *init*;

Description

define atomic-variable defines module variables in the current module. These variables have all the properties of normal module variables, and in addition they may be atomically tested and updated with **conditional-update**!.

Example

define atomic-variable *number-detected* = 0;

synchronized-class [Definition]

Syntax

As for **define class**, and permits the additional slot adjective **atomic.**

Description

define synchronized-class is upwards compatible with **define class**, exported from the **dylan** library. It provides an additional option which permits the definition of slots which may be conditionally updated. It is undefined whether the module variables **class-definer** exported from the **dylan** library and **synchronized-class-definer** exported from the **threads** library are the same module variable.

The **atomic** adjective may be used to define conditional-updater methods for individual slots defined by **define synchronized-class**. This adjective may only be used for slots with **instance** allocation. The adjective causes the additional creation of a conditional-updater method for the slot, suitable for use with the **conditional-update!** macro.

A conditional-updater method has three required parameters **(new-value, old-value, object)**. The method atomically compares the slot contents for **object** with **old-value**, and conditionally updates the slot. If the comparison fails, the method returns **#f**. Otherwise, the slot contents are replaced with **new-value**, and the method returns a true value.

Keywords

The following additional keywords are permitted for **atomic** slots:

conditional-updater:
The name of a module variable to which the conditional-updater method should be added, or **#f** if no conditional-updater method should be defined (in which case the use of the **atomic** adjective was unnecessary). If it is not supplied, it will default to **getter-name-conditional-updater**, where **getter-name** names the getter for the slot.

Example

The following example defines a class with a single slot suitable for conditional updating:

```
define synchronized-class <atomic-value> (<object>)
    atomic slot counter :: <integer>
end class
```

This example defines a conditional-updater method with the following signature:

```
define method counter-conditional-updater
        (new-value, old-value, object :: <atomic-value>)
end method
```

conditional-update! [Macro]

Syntax

conditional-update!(*local-name* = *place***)**
    body
[**success** *success-expr*]
[**failure** *failure-expr*]
end

Description

The value of the place is evaluated once to determine the initial value, which is then bound to the *local-name* as a lexical variable. The body is then evaluated to determine the new value for the place. The place is then conditionally updated — which means that the following steps are performed atomically:

1. The place is evaluated again and compared with the initial value in *local-name* using \==
2. If the value was found to be the same as the initial value, then the new value is stored by assignment, otherwise the conditional update fails.

If the update was successful, then **conditional-update!** returns the result of the **success** expression, or returns the new value of the place if no **success** clause was supplied.

If the update failed, then **conditional-update!** signals a condition, unless a **failure** clause was given, in which case the value is returned.

If the *place* is a *name*, it must be the name of an **atomic-variable** in the current module scope. If *place* is not a name, then it may have the syntax of a call to a function. This permits an *extended form* for **conditional-update!**, by analogy with the extended form for **:=**. In this case, if the place appears syntactically as **name(arg1, ...argn)**, then the macro expands into a call to the function **name-conditional-updater(***new-value***,** *local-name***, arg1, ... argn)**. If the result of this function call is **#f**, then the conditional update is deemed to have failed.

Exceptions

conditional-update! may signal a condition of the following class (which is a subclass of **<error>**), unless a **failure** clause is supplied.

<conditional-update-error>

Example

The following example does an atomic increment of ***number-detected***.

```
until (conditional-update! (current-val = *number-detected*)
                current-val + 1
            failure #f
        end conditional-update!)
end until
```

3.4.7 Fluid Binding

fluid-variable [Definition]

Syntax

define fluid-variable *bindings* = *init***;**

Description

define fluid-variable defines module variables in the current module which have bindings in the dynamic environment. The initialization expression is evaluated once, and is used to provide the initial values for the implicit bindings in the dynamic environment of each thread. These bindings are therefore thread-local. The bindings may be dynamically shadowed with **fluid-bind**. The value of a fluid-variable binding may be changed with the normal assignment operator **:=**. There is currently no way to share bindings, either temporarily or permanently between one or more threads, short of using a normal variable.

Example

```
define fluid-variable *standard-output*
    = make(<standard-output-stream>);
```

fluid-bind [Macro]

Syntax

fluid-bind (*place1* = *init1***,** *place2* = *init2***, ...)** *body* **end;**

Description

The body is evaluated in an environment in which the specified *places* are rebound in the dynamic environment, and initialized to the results of evaluating the initialization expressions.

If the *place* is a *name*, it must be the name of a **fluid-variable** in the module scope.

If *place* is not a name, then it may have the syntax of a call to a function. This permits an *extended form* for **fluid-bind**, by analogy with the extended form for **:=**. In this case, if the place appears syntactically as **name(arg1, ...argn)**, then the macro expands into a call to the function **name-fluid-binder(***init***,** *body-method***, arg1, ... argn)** where *init* is the initial value for the binding, and *body-method* is function with no parameters whose body is the *body of* the **fluid-bind**.

There are no features in the current version of the threads library which make use of the extended form of **fluid-bind**.

Example

The following example shows the simple form of **fluid-bind**.

```
fluid-bind (*standard-output* = new-val())
    top-level-loop ()
end;
```

Example 2

The following example shows the extended form of **fluid-bind**.

```
fluid-bind (object.a-slot = new-slot-val())
    inner-body(object)
end;
```

This expands into code equivalent to the following:

```
a-slot-fluid-binder( new-slot-val(),
                     method () inner-body(object) end,
                     object)
```

3.5 Thread Safety in Other Libraries

3.5.1 General Requirements

Library designers are responsible for documenting which features of the library offer in-built synchronization and which do not. There is no definitive rule to guide designers on how to do this, but the following guidelines may be useful.

If a client of the library forgets to use a synchronization feature when one is necessary, then the library designer should ensure that the effect of the lack of synchronization is limited to a small unit (probably a single object). In cases where the designer cannot guarantee that the effect will be limited, the designer should either define that the library will implement the synchronization internally, or will provide a function or macro for clients to use instead.

Library implementors must ensure that the library provides implicit synchronization for any hidden global state which is maintained by the library. Library designers may choose whether the library should offer implicit synchronization of the state of objects managed by the library. The interface is more convenient if the synchronization is implicit, but it may be more efficient to rely on explicit synchronization by the client. Library designers should always document the choice they make.

3.5.2 The Dylan Library

The definition of the Dylan library is not changed with the addition of the threads library. The implementation will ensure that all hidden global state (such as the symbol table and any generic function caches) is implicitly synchronized. Those functions in the Dylan library which are defined to modify the state of objects are not defined to provide implicit synchronization. However, implementations are expected to provide enough implicit synchronization to ensure that the effects of an omission of explicit synchronization are sufficiently limited that language integrity and runtime safety are not violated.

4. Portability Support for Dylan Threads

4.1 Dylan Portability Interface

The Simple Threads Library is designed for implementation using different threads APIs from common operating systems, including Solaris, Win32 and MacOS. Harlequin's implementation of the library is designed to be directly portable onto these operating systems. This portability is achieved by using ANDF tokens to support primitive operations defined within our implementation of the library. Each primitive operation must be implemented specially for each operating system. This can be achieved by having an operating system specific token expansion, or, alternatively, by portably mapping the token onto a call to a function in Dylan's runtime system. The Dylan runtime system is implemented specially for each platform.

The set of portable primitive operations which correspond to ANDF tokens is collectively called the *portability layer*. The back end of the Dylan compiler has special knowledge of the portability layer. This is achieved by having code generation methods for the ANDF back end which are specialized on nodes in the compiler flow graph that correspond to the primitive operations. Some of these nodes appear to the compiler as calls to *primitive functions* (which are treated specially by the back end). Other nodes describe the operation more directly in the flow graph. The specialized methods emit uses of the corresponding ANDF tokens.

4.1.1 Portability and Runtime Layers

The design assumes that each of the concrete classes of the Simple Threads Library (**<thread>**, **<simple-lock>**, **<recursive-lock>**, **<semaphore>** and **<notification>**) corresponds with an equivalent lower-level feature provided directly by either the operating system or the runtime system. The Dylan objects which are instances of these classes are implemented as *containers* for handles corresponding to low-level (non-Dylan) objects. The Dylan objects contain normal Dylan slots too, and these are directly manipulated by the Dylan library. However, the slots containing the low-level handles may only be manipulated via primitive function calls. For each of the classes, primitive functions are defined to both create and destroy the low-level handles, as well as to perform the basic functions of the class, such as **wait-for** and **release**. The platform-specific implementation of these primitive functions is free to choose any representation for these handles, provided that it is the same shape as a Dylan slot (which is equivalent to C's **void ***).

As with all Dylan objects, the container objects defined by the threads library are subject to automatic memory management, and possible relocation by the garbage collector. The contents of the container slots will be copied during such a relocation — but the values they contain will not be subject to garbage collection or relocation themselves.

The portability layer provides no direct support for the **fluid-bind** operation. The library implements a **fluid-variable** as a thread-local variable, and uses the high-level Dylan construct **unwind-protect** [also called **cleanup** in Dylan's infix syntax] to manage the creation and deletion of new bindings.

The portability layer includes support for conditional update of atomic variables, as well as assignment. The implementation mechanism for these is not defined, but it is hoped that many platforms will provide direct hardware support for this operation. Where hardware support is not available, the low-level implementation may choose to use a lock to protect conditional updates and assignments, as a fall back option. It is assumed that atomic variables may always be read as normal variables.

Table 1 on page 24 shows the expected mapping between the concrete Dylan classes and low-level operating system features, for three of the most popular general-purpose operating systems. In principle, there exist separate expansions of ANDF tokens and separate implementations of primitive functions for each operating system. This document describes in detail only the Solaris implementation.

Table 1. Implementations of Dylan Thread Interfaces

Dylan Interface	**Solaris Implementation**	**Win32 Implementation**	**Macintosh Implementation**
<thread>	thread	thread	thread
<simple-lock>	mutex	critical region	critical region
<recursive-lock>	mutex	critical region	critical region
<semaphore>	semaphore	semaphore	built using critical region as a primitive
<notification>	condition variable	event	built using critical region as a primitive
fluid-variable	thread-local variable	thread-local variable	thread-local variable?
conditional-update!	mutex	exchange instruction (using a guard value as a lock);	store conditional instruction (on Power PC)

4.1.2 Dylan Types for Threads Portability

Three Dylan types merit discussion for their use with portability primitives: **<thread>**, **<portable-container>**, and **<optional-name>**. Objects that are instances of the **<thread>** and **<portable-container>** classes have slots which contain lower-level objects that are specific to the Dylan runtime or operating system. The **<optional-name>** type allows an object, such as a lock, to have a name represented as a string or, if no name is supplied, as the Boolean false value **#f**.

<thread> [Class]

A Dylan object of class **<thread>** contains two OS handles. One of these represents the underlying OS thread, and the other may be used by implementations to contain the current status of the thread, as an aid to the implementation of the join state.

<portable-container> [Class]

The **<portable-container>** class is used by the implementation as a superclass for all the concrete synchronization classes (**<simple-lock>**, **<recursive-lock>**, **<semaphore>**, and **<notification>**). Each **<portable-container>** object contains an OS handle, which is available to the runtime for storing any OS-specific data. Subclasses may provide additional slots.

<optional-name> [Type]

This is a union type which is used to represent names of synchronization objects. Values of the type are either strings (of class **<byte-string>**) or false (**#f**).

Various classes of Dylan objects are passed through the portability interface, and hence require description in terms of lower level languages. Table 2 on page 25 maps the layout of these Dylan objects onto their C equivalents, which are used by runtime-specific implementations of the portability layer.

In general, all Dylan types can be thought of as equivalent to the C type **Z**, which is in turn equivalent to the C type **void***. Of course, runtime-specific implementations of the portability layer must have access to relevant fields of the Dylan objects on which they operate. The type definitions in Table 2 give implementations access to fields needed for specific types. These definitions are not necessarily complete descriptions of the Dylan objects, however. The objects may contain additional fields that are not of interest to the portability layer, and subclasses may add additional fields of their own.

Table 2. Correspondence Between Dylan Types and C Types

Dylan Type	**C Type**	**C Type Definition**
<object>	**Z**	**typedef void* Z;**
<small-integer>	**ZINT**	**platform specific (size of void*)**
<function>	**ZFN**	**typedef Z(*ZFN)(Z, int, …);**

<simple-object-vector>	**SOV***	**typedef struct _sov {
		Z class;
		ZINT size;
		Z data[ ];
		} SOV;**

<byte-string>	**B_STRING***	**typedef struct _bst {
		Z class;
		ZINT size;
		char data[ ];
		} B_STRING;**

<optional-name>	**D_NAME**	**typedef void* D_NAME;**

<portable-container>	**CONTAINER***	**typedef struct _ctr {
		Z class;
		void* handle;
		} CONTAINER;**

<thread>	**D_THREAD***	**typedef struct _dth {
		Z class;
		void* handle1;
		void* handle2;
		} D_THREAD;**

4.2 Compiler Support for the Portability Interface

4.2.1 The Compiler Flow Graph

The front end of the compiler parses Dylan source code and produces an intermediate representation, the Implicit Continuation Representation (ICR). The ICR is a directed acyclic graph (DAG) of Dylan objects.

A *leaf* in the ICR represents a basic computational object, such as a variable (of class **<variable-leaf>**) or a function (of class **<function-leaf>**). A *node* in the ICR represents an operation such as assignment (class **<assignment>**), conditional execution (class **<if>**), or a reference to a leaf (class **<reference>**).

In mapping Dylan code to the ICR, the compiler uses a set of *converters*, which perform syntactic pattern matching against fragments of Dylan code and generate the ICR corresponding to the matched code. For example, when the compiler encounters a top-level variable definition (introduced by the Dylan **define variable** construct), the converter for **define variable** creates a new instance of **<global-variable-leaf>** in the ICR to represent this variable and to record data such as its name, initial value, and typing information.

The back end of the compiler traverses the flow graph and emits code in the target language for compiler output. Methods in the back end specialize on node and leaf classes to enable them to produce the appropriate output. Some methods in the ANDF back end, for example, are specialized on node classes representing calls to primitive functions. These methods emit uses of the ANDF tokens that correspond to the function calls.

4.2.2 Compiler Support for Atomic and Fluid Variables

The portability layer provides support for atomic variable access and for Dylan fluid variables (implemented as thread-local variables). Atomic variables and thread variables are directly represented in the flow graph, where they are subject to dataflow analysis. The variables themselves appear as leaves in the graph.

Because both atomic and fluid variables need special treatment when they are accessed, the back end must emit output that is different from that for accessing other kinds of variables. The compiler defines two specialized classes of leaf for the ICR, **<atomic-global-variable-leaf>** (corresponding to atomic variables) and **<fluid-global-variable-leaf>** (corresponding to fluid variables). These are subclasses of **<global-variable-leaf>** and therefore inherit general characteristics of leaves that represent variables.

ICR leaves representing both atomic and fluid variables are created by the converter for **define variable**. When the compiler encounters a definition of an atomic variable (introduced by the **define atomic-variable** construct), the converter for **define variable** creates an instance of **<atomic-global-variable-leaf>** in the ICR. When the compiler encounters a definition of a fluid variable (introduced by the **define fluid-variable** construct), the converter creates an instance of **<fluid-global-variable-leaf>**.

The operations of reading, writing, and conditionally updating atomic variables and of reading and writing fluid variables are not represented by primitive functions. Instead, they are represented directly in the flow graph. By specializing methods on the leaf classes that represent atomic and fluid variables, the ANDF back end generates uses of ANDF tokens that correspond to these operations.

4.2.3 Compiler Support for Primitives

When the compiler constructs the flow graph, it represents a function call as a node in the ICR. Just as the compiler distinguishes atomic and fluid variables by means of specialized leaf classes, so it distinguishes calls to primitive functions of the portability interface by means of a specialized node class.

A function call is an operation on several components: the function object, the arguments, and the destination for returned values. When the compiler encounters a regular Dylan call, which typically appears as a call to a generic function, it represents the call in the ICR as a node of class **<combination>**.

However, the compiler contains a table of the primitive functions in the portability interface. Before creating an ICR node to represent a function call, the compiler looks up the function being called in the table of primitives. If the function appears in the table, the compiler creates an ICR node of class **<primitive-combination>**.

When the ANDF back end traverses the flow graph, methods specialized on the node class **<primitive-combination>** emit uses of the ANDF tokens that correspond to calls to primitive functions.

4.3 Primitive Functions of the Portability Interface

This section describes in detail the arguments, values, and operations of the primitive functions.

4.3.1 Threads

primitive-make-thread [Primitive]

Signature

**(thread :: <thread>, name :: <optional-name>, priority :: <small-integer>,
function :: <function>) => ()**

Arguments

thread	A Dylan thread object.
name	The name of the thread (as a **<byte-string>**) or **#f**.
priority	The priority at which the thread is to run.
function	The initial function to run after the thread is created.

Description

Creates a new OS thread and destructively modifies the container slots in the Dylan thread object with the handles of the new OS thread. The new OS thread is started in a way which calls the supplied Dylan function.

primitive-destroy-thread [Primitive]

Signature

(thread :: <thread>) => ()

Arguments

thread	A Dylan thread object.

Description

Frees any runtime-allocated memory associated with the thread.

primitive-initialize-current-thread [Primitive]

Signature

(thread :: <thread>) => ()

Arguments

thread	A Dylan thread object.

Description

The container slots in the Dylan thread object are destructively modified with the handles of the current OS thread. This function will be used to initialize the first thread, which will not have been started as the result of a call to **primitive-make-thread**.

primitive-thread-join-single [Primitive]

Signature

(thread :: <thread>) => (error-code :: <small-integer>)

Arguments

thread	A Dylan thread object.

Values

error-code	0 = ok, anything else is an error, corresponding to a multiple join.

Description

The calling thread blocks (if necessary) until the specified thread has terminated.

primitive-thread-join-multiple [Primitive]

Signature

(thread-vector :: <simple-object-vector>) => (result)

Arguments

thread-vector	A **<simple-object-vector>** containing **<thread>** objects

Values

result	The **<thread>** that was joined, if the join was successful; otherwise, a **<small-integer>** indicating the error.

Description

The calling thread blocks (if necessary) until one of the specified threads has terminated.

primitive-thread-yield [Primitive]

Signature

() => ()

Description

For co-operatively scheduled threads implementations, the calling thread yields execution in favour of another thread. This may do nothing in some implementations.

primitive-current-thread [Primitive]

Signature

() => (thread-handle)

Values

thread-handle	A low-level handle corresponding to the current thread

Description

Returns the low-level handle of the current thread, which is assumed to be in the handle container slot of one of the **<thread>** objects known to the Dylan library. This result is therefore NOT a Dylan object. The mapping from this value back to the **<thread>** object must be performed by the Dylan threads library, and not the primitive layer, because the **<thread>** object is subject to garbage collection, and may not be referenced from any low-level data structures.

4.3.2 Simple Locks

primitive-make-simple-lock [Primitive]

Signature

(lock :: <portable-container>, name :: <optional-name>) => ()

Arguments

lock	A Dylan **<simple-lock>** object.
name	The name of the lock (as a **<byte-string>**) or **#f**.

Description

Creates a new OS lock and destructively modifies the container slot in the Dylan lock object with the handle of the new OS lock.

primitive-destroy-simple-lock [Primitive]

Signature

(lock :: <portable-container>) => ()

Arguments

lock	A Dylan **<simple-lock>** object.

Description

Frees any runtime-allocated memory associated with the lock.

primitive-wait-for-simple-lock [Primitive]

Signature

(lock :: <portable-container>) => (error-code :: <small-integer>)

Arguments

lock	A Dylan **<simple-lock>** object.

Values

error-code	0 = ok

Description

The calling thread blocks until the specified lock is available (unlocked) and then locks it. When the function returns, the lock is owned by the calling thread.

primitive-wait-for-simple-lock-timed [Primitive]

Signature

(lock :: <portable-container>, millisecs :: <small-integer>)
=> (error-code :: <small-integer>)

Arguments

lock	A Dylan **<simple-lock>** object.
millisecs	Timeout period in milliseconds

Values

error-code	0 = ok, 1 = timeout expired

Description

The calling thread blocks until either the specified lock is available (unlocked) or the timeout period expires. If the lock becomes available, this function locks it. If the function returns 0, the lock is owned by the calling thread, otherwise a timeout occurred.

primitive-release-simple-lock [Primitive]

Signature

(lock :: <portable-container>) => (error-code :: <small-integer>)

Arguments

lock	A Dylan **<simple-lock>** object.

Values

error-code	0 = ok, 2 = not locked

Description

Unlocks the specified lock. The lock must be owned by the calling thread, otherwise the result indicates “not locked”.

primitive-owned-simple-lock [Primitive]

Signature

(lock :: <portable-container>) => (owned :: <small-integer>)

Arguments

lock	A Dylan **<simple-lock>** object.

Values

owned	0= not owned, 1 = owned

Description

Returns 1 if the specified lock is owned (locked) by the calling thread.

4.3.3 Recursive Locks

primitive-make-recursive-lock [Primitive]

Signature

(lock :: <portable-container>, name :: <optional-name>) => ()

Arguments

lock	A Dylan **<recursive-lock>** object.
name	The name of the lock (as a **<byte-string>**) or **#f**.

Description

Creates a new OS lock and destructively modifies the container slot in the Dylan lock object with the handle of the new OS lock.

primitive-destroy-recursive-lock [Primitive]

Signature

(lock :: <portable-container>) => ()

Arguments

lock	A Dylan **<recursive-lock>** object.

Description

Frees any runtime-allocated memory associated with the lock.

primitive-wait-for-recursive-lock [Primitive]

Signature

(lock :: <portable-container>) => (error-code :: <small-integer>)

Arguments

lock	A Dylan **<recursive-lock>** object.

Values

error-code	0 = ok

Description

The calling thread blocks until the specified lock is available (unlocked or already locked by the calling thread). When the lock becomes available, this function claims ownership of the lock and increments the lock count. When the function returns, the lock is owned by the calling thread.

primitive-wait-for-recursive-lock-timed [Primitive]

Signature

(lock :: <portable-container>, millisecs :: <small-integer>)
    => (error-code :: <small-integer>)

Arguments

lock	A Dylan **<recursive-lock>** object.
millisecs	Timeout period in milliseconds

Values

error-code	0 = ok, 1 = timeout expired

Description

The calling thread blocks until the specified lock is available (unlocked or already locked by the calling thread). If the lock becomes available, this function claims ownership of the lock, increments an internal lock count, and returns 0. If a timeout occurs, the function leaves the lock unmodified and returns 1.

primitive-release-recursive-lock [Primitive]

Signature

(lock :: <portable-container>) => (error-code :: <small-integer>)

Arguments

lock	A Dylan **<recursive-lock>** object.

Values

error-code	0 = ok, 2 = not locked

Description

Checks that the lock is owned by the calling thread, and returns 2 if not. If the lock is owned, its internal count is decremented by 1. If the count is then zero, the lock is then released.

primitive-owned-recursive-lock [Primitive]

Signature

(lock :: <portable-container>) => (owned :: <small-integer>)

Arguments

lock	A Dylan **<recursive-lock>** object.

Values

owned	0= not owned, 1 = owned

Description

Returns 1 if the specified lock is locked and owned by the calling thread.

4.3.4 Semaphores

primitive-make-semaphore [Primitive]

Signature

**(lock :: <portable-container>, name :: <optional-name>,
initial :: <small-integer>, max :: <small-integer>) => ()**

Arguments

lock	A Dylan **<semaphore>** object.
name	The name of the lock (as a **<byte-string>**) or **#f**.
initial	The initial value for the semaphore count

Description

Creates a new OS semaphore with the specified initial count and destructively modifies the container slot in the Dylan lock object with the handle of the new OS semaphore.

primitive-destroy-semaphore [Primitive]

Signature

(lock :: <portable-container>) => ()

Arguments

lock	A Dylan **<semaphore>** object.

Description

Frees any runtime-allocated memory associated with the semaphore.

primitive-wait-for-semaphore [Primitive]

Signature

(lock :: <portable-container>) => (error-code :: <small-integer>)

Arguments

lock	A Dylan **<semaphore>** object.

Values

error-code	0 = ok

Description

The calling thread blocks until the internal count of the specified semaphore becomes greater than zero. It then decrements the semaphore count.

primitive-wait-for-semaphore-timed [Primitive]

Signature

(lock :: <portable-container>, millisecs :: <small-integer>)
=> (error-code :: <small-integer>)

Arguments

lock	A Dylan **<semaphore>** object.
millisecs	Timeout period in milliseconds

Values

error-code	0 = ok, 1 = timeout expired

Description

The calling thread blocks until either the internal count of the specified semaphore becomes greater than zero or the timeout period expires. In the former case, the function decrements the semaphore count and returns 0. In the latter case, the function returns 1.

primitive-release-semaphore [Primitive]

Signature

(lock :: <portable-container>) => (error-code :: <small-integer>)

Arguments

lock	A Dylan **<semaphore>** object.

Values

error-code	0 = ok, 3 = count exceeded

Description

This function checks that internal count of the semaphore is not at its maximum limit, and returns 3 if the test fails. Otherwise the internal count is incremented.

4.3.5 Notifications

primitive-make-notification [Primitive]

Signature

(notification :: <portable-container>, name :: <optional-name>) => ()

Arguments

notification	A Dylan <**notification>** object.
name	The name of the notification (as a **<byte-string>**) or **#f**.

Description

Creates a new OS notification (condition variable) and destructively modifies the container slot in the Dylan lock object with the handle of the new OS notification.

primitive-destroy-notification [Primitive]

Signature

(notification :: <portable-container>) => ()

Arguments

notification	A Dylan **<notification>** object.

Description

Frees any runtime-allocated memory associated with the notification.

primitive-wait-for-notification [Primitive]

Signature

(notification :: <portable-container>, lock :: <portable-container>)
    => (error-code :: <small-integer>)

Arguments

notification	A Dylan **<notification>** object.
lock	A Dylan **<simple-lock>** object.

Values

error-code	0 = ok, 2 = not locked, 3 = other error

Description

The function checks that the specified lock is owned by the calling thread, and returns 2 if the test fails. Otherwise, the calling thread atomically releases the lock and then blocks, waiting to be notified of the condition represented by the specified notification. When the calling thread is notified of the condition, the function reclaims ownership of the lock, blocking if necessary, before returning 0.

primitive-wait-for-notification-timed [Primitive]

Signature

**(notification :: <portable-container>, lock :: <portable-container>,
millisecs :: <small-integer>) => (error-code :: <small-integer>)**

Arguments

notification	A Dylan **<notification>** object.
lock	A Dylan **<simple-lock>** object.
millisecs	Timeout period in milliseconds

Values

error-code	0 = ok, 1 = timeout, 2 = not locked, 3 = other error

Description

The function checks that the specified lock is owned by the calling thread, and returns 2 if the test fails. Otherwise, the calling thread atomically releases the lock and then blocks, waiting to be notified of the condition represented by the specified notification, or for the timeout period to expire. The function then reclaims ownership of the lock, blocking indefinitely if necessary, before returning either 0 or 1 to indicate whether a timeout occurred.

primitive-release-notification [Primitive]

Signature

**(notification :: <portable-container>, lock :: <portable-container>)
=> (error-code :: <small-integer>)**

Arguments

notification	A Dylan **<notification>** object.
lock	A Dylan **<simple-lock>** object.

Values

error-code	0 = ok, 2 = not locked

Description

If the calling thread does not own the specified lock, the function returns the error value 2. Otherwise, the function releases the specified notification, notifying another thread that is blocked waiting for the notification to occur. If more than one thread is waiting for the notification, it is unspecified which thread is notified. If no threads are waiting, then the release has no effect.

primitive-release-all-notification [Primitive]

Signature

(notification :: <portable-container>, lock :: <portable-container>)
=> (error-code :: <small-integer>)

Arguments

notification	A Dylan **<notification>** object.
lock	A Dylan **<simple-lock>** object.

Values

error-code	0 = ok, 2 = not locked

Description

If the calling thread does not own the specified lock, the function returns the error value 2. Otherwise, the function releases the specified notification, notifying all other threads that are blocked waiting for the notification to occur. If no threads are waiting, then the release has no effect.

4.3.6 Timers

primitive-sleep [Primitive]

Signature

(millisecs :: <small-integer>) => ()

Arguments

millisecs	Time interval in milliseconds

Description

This function causes the calling thread to block for the specified time interval.

4.3.7 Thread Variables

primitive-allocate-thread-variable [Primitive]

Signature

(initial-value) => (handle-on-variable)

Arguments

initial-value	A Dylan object that is to be the initial value of the fluid variable.

Values

handle-on-variable

An OS handle on the fluid variable, to be stored as the immediate value of the variable. Variable reading and assignment will indirect through this handle. The handle is not a Dylan object.

Description

This function creates a new thread-local variable handle, and assigns the specified initial value to the location indicated by the handle. The function must arrange to assign the initial value to the thread-local location associated with all other existing threads, too. The function must also arrange that whenever a new thread is subsequently created, it also has its thread-local location indicated by the handle set to the initial value.

4.4 Support for Dylan Language Features

4.4.1 Garbage Collector Support for Threads

There are some general constraints that any combination of a threads implementation and a garbage collector implementation must agree about. A detailed discussion is given in [ME94]. One important consideration is that the garbage collector must be able to locate all the roots of the entire program — including local variables on the stack, and thread-local variables.

With Harlequin’s design for the Dylan threads library, it is the responsibility of the runtime system to perform the allocation of the stacks for each thread, as well as the allocation of the static thread-local variables. Since the runtime system is also responsible for providing garbage collection services, co-operation between the threads and memory management subsystems may be achieved without any active support from the ANDF output produced from Dylan code. This allows the runtime system implementor considerable freedom to select the most appropriate designs for each platform.

Harlequin’s Simple Threads Library has been tested on Solaris with a public domain conservative garbage collector [BW88]. This collector has implicit support for locating the stacks of multiple threads, and for tracing memory which is allocated with **malloc**. The thread-local variables used by Dylan are found by the Boehm collector because they are ultimately located in memory reserved with **malloc** by the threads runtime system, with **primitive_allocate_thread_variable**. The Boehm collector also accepts direct responsibility for synchronization of all allocation requests.

Ultimately, Harlequin plan to use a more sophisticated *mostly copying* collector for the Dylan threads library. This work is outside of the scope of the GLUE project, but it is believed that the interface to the runtime system defined here is sufficiently flexible that such an implementation is possible without any significant change to the producer output.

4.4.2 Interfacing to Foreign Code

It is intended that threads created by the Dylan library may inter-operate with code written in other languages with no special constraints. The issues relating to inter-operation of Dylan with other languages were discussed in [MG95], but limited to a serial computation model. It is relatively straightforward to extend the conclusions of this discussion to support multiple threads.

Dylan may be interfaced with other languages via a Foreign Language Interface (FLI) [MG95], which acts as a barrier between Dylan conventions and the *neutral* conventions of the platform. The FLI is responsible for:

1. mapping between Dylan and foreign data types,
2. converting between Dylan and foreign calling conventions
3. maintaining the Dylan dynamic environment
4. maintaining any support necessary for garbage collection (such as ensuring that all Dylan values can be traced).

The first and second of these require no significant extensions to support multiple threads, since these are inherently computations which have no effect on any thread other than the one performing the computation.

As discussed in [ME94], there is a requirement that the dynamic environment for each thread is stored in a thread-local variable. It is possible to implement this variable, and all references to it, entirely within the runtime system, so this does not place a requirement on ANDF to support thread-local variables. Since the environment is stored in this way, its value is preserved across calls into foreign code, and it will still be valid if the foreign code calls back into Dylan. The techniques described in [MG95] for maintaining the dynamic environment across foreign calls are therefore directly appropriate to a multi-threaded implementation too.

[ME94] discusses the general issues of support for garbage collection in a multi-threaded system, and shows how garbage collector techniques may be implemented in the Dylan runtime system to manage the Dylan object space. [MG95] describes how the garbage collector may require an *interface record* which records those objects passed from Dylan to foreign code, so that the garbage collector will treat them as both live and static. This technique may be extended directly for use with multiple threads.

If an object is passed to foreign code with dynamic extent, then it is sufficient to ensure that the object is referenced from the current stack, which the garbage collector will scan conservatively. In a multi-threaded implementation, the garbage collector will scan all the stacks conservatively, so there is no requirement to maintain a thread-global data structure.

If an object is passed with indefinite extent, then it must be recorded in a table. The table may be maintained by the runtime system, by means of suitable primitive functions to add and remove references. There are potentially synchronization problems associated with multiple threads manipulating a global data structure — but the runtime system implementation is free to choose whether to have separate tables for each thread, or whether to have a global table with an associated lock to guard accesses. Either technique is possible — but Harlequin have not yet implemented this feature.

One further consideration is the interaction of the Dylan threads library itself with foreign components:

If foreign code is not designed for multiple threads (for instance, because it uses global data structures, and doesn't synchronize updates), then the code may fail if it is invoked from multiple Dylan threads. However, this problem is not related to the Dylan implementation, since it would fail if called from multiple threads created by any means. The solution is to modify the foreign component to make it thread safe.

If foreign code is designed for use with multiple threads, then it is valid for it to use the synchronization facilities of the Dylan library (by calling back into Dylan, to invoke the Simple Threads Library synchronization functions). Alternatively, it may use its own methods for synchronization, provided that these are not incompatible with the methods provided by the operating system. This is valid whenever it has been possible to implement the runtime system support for threads directly in terms of operating system features, and it is anticipated that this will always be true if the operating system supports threads. Typically, foreign code is expected to make direct use of operating system threads facilities.

However, a problem may arise if a thread is created in foreign code, and the new thread then calls back into Dylan. In this case, the Dylan thread library itself will not be able to find an existing **<thread>** object corresponding to the current thread, and the fluid variables for the current thread will not have been correctly initialized. Worse still, the garbage collector may not have enough information to locate the roots of the thread. Harlequin have not yet allowed for this in their implementation, but they have an anticipated solution.

It is possible to detect that a thread has never been executing on the Dylan side of the FLI before because it will have an uninitialized (zero) value for its thread-local dynamic environment variable. This can be checked at a call-in in the stub function which implements the FLI. Once such a thread has been detected, appropriate initialization steps can be taken. A function in the runtime system can be called to register the stack of the thread for root tracing; the dynamic environment can be set to point to a suitable value on the stack; finally a new Dylan **<thread>** object can be allocated and initialized with **primitive-initialize-current-thread** (as for the first thread).

4.4.3 Finalization

As has been discussed, the Dylan synchronization objects are implemented as wrappers around lower-level operating system structures. The Dylan objects are subject to garbage collection, and their memory will be automatically freed by the garbage collector at an undefined point in the program. But the low-level structures are not Dylan objects and must be explicitly freed when the Dylan container is collected (primitive functions are provided for this purpose). However, the core language of Dylan provides no *finalization* mechanism to invoke cleanup code when objects are reclaimed. Harlequin's implementation of the Simple Threads Library strictly requires this, but it is not yet implemented. It is intended to provide finalization support for Dylan with a new garbage collector which is currently under development.

5. ANDF Token Design

The ANDF compiler back end supports Dylan threads entirely through tokens. The use of tokens allows for operating system specific implementation by providing either specific token expansions or specific implementations of the primitive functions invoked by the tokens.

The compiler back end operates on a straight-forward mapping between Dylan primitives and ANDF tokens. Each Dylan primitive has an associated ANDF token, as specified in Table 3 on page 42. Code-generation methods in the ANDF back end are specialized on classes of flow-graph nodes that represent calls to Dylan primitives. When the back end encounters a call to a Dylan primitive, it emits a use of the corresponding token.

Table 3. Correspondence Between Dylan Primitives and ANDF Tokens

Dylan Primitive	**ANDF Token**
primitive-make-thread	PRIM_MAKE_THREAD
primitive-destroy-thread	PRIM_DESTROY_THREAD
primitive-initialize-current-thread	PRIM_INITIALIZE_CURRENT_THREAD
primitive-thread-join-single	PRIM_THREAD_JOIN_SINGLE
primitive-thread-join-multiple	PRIM_THREAD_JOIN_MULTIPLE
primitive-thread-yield	PRIM_THREAD_YIELD
primitive-current-thread	PRIM_CURRENT_THREAD
primitive-make-simple-lock	PRIM_MAKE_SIMPLE_LOCK
primitive-destroy-simple-lock	PRIM_DESTROY_SIMPLE_LOCK
primitive-wait-for-simple-lock	PRIM_WAIT_FOR_SIMPLE_LOCK
primitive-wait-for-simple-lock-timed	PRIM_WAIT_FOR_SIMPLE_LOCK_TIMED
primitive-release-simple-lock	PRIM_RELEASE_SIMPLE_LOCK
primitive-owned-simple-lock	PRIM_OWNED_SIMPLE_LOCK
primitive-make-recursive-lock	PRIM_MAKE_RECURSIVE_LOCK
primitive-destroy-recursive-lock	PRIM_DESTROY_RECURSIVE_LOCK
primitive-wait-for-recursive-lock	PRIM_WAIT_FOR_RECURSIVE_LOCK
primitive-wait-for-recursive-lock-timed	PRIM_WAIT_FOR_RECURSIVE_LOCK_TIMED
primitive-release-recursive-lock	PRIM_RELEASE_RECURSIVE_LOCK
primitive-owned-recursive-lock	PRIM_OWNED_RECURSIVE_LOCK
primitive-make-semaphore	PRIM_MAKE_SEMAPHORE
primitive-destroy-semaphore	PRIM_DESTROY_SEMAPHORE
primitive-wait-for-semaphore	PRIM_WAIT_FOR_SEMAPHORE
primitive-wait-for-semaphore-timed	PRIM_WAIT_FOR_SEMAPHORE_TIMED
primitive-release-semaphore	PRIM_RELEASE_SEMAPHORE
primitive-make-notification	PRIM_MAKE_NOTIFICATION
primitive-destroy-notification	PRIM_DESTROY_NOTIFICATION
primitive-wait-for-notification	PRIM_WAIT_FOR_NOTIFICATION
primitive-wait-for-notification-timed	PRIM_WAIT_FOR_NOTIFICATION_TIMED
primitive-release-notification	PRIM_RELEASE_NOTIFICATION
primitive-release-all-notification	PRIM_RELEASE_ALL_NOTIFICATION

Dylan Primitive	ANDF Token
primitive-sleep	PRIM_SLEEP
primitive-allocate-thread-variable	PRIM_ALLOCATE_THREAD_VARIABLE

The signature of each token corresponds directly to the signature of the corresponding Dylan primitive. In the token signatures, all parameters and return values are of sort EXP. The token signatures make use of two shapes, pz and ~ptr-void.

pz [Shape]

This shape represents any Dylan object.

~ptr_void [Shape]

This shape represents the C type **void***. It is defined in the standard ANSI C token library.

5.1 ANDF Tokens Corresponding to Primitives

5.1.1 Threads

PRIM_MAKE_THREAD [Token]

Signature

thread_object:	EXP pz
name:	EXP pz
priority:	EXP pz
function:	EXP pz
$\rightarrow$	EXP pz

PRIM_DESTROY_THREAD [Token]

Signature

thread_handle:	EXP pz
$\rightarrow$	EXP pz

PRIM_INITIALIZE_CURRENT_THREAD [Token]

Signature

thread:	EXP pz
$\rightarrow$	EXP pz

PRIM_THREAD_JOIN_SINGLE [Token]

Signature

thread_handle:	EXP pz
$\rightarrow$	EXP pz

PRIM_THREAD_JOIN_MULTIPLE [Token]

Signature

thread_list:	EXP pz
$\rightarrow$	EXP pz

PRIM_THREAD_YIELD [Token]

Signature

$\rightarrow$	EXP pz

PRIM_CURRENT_THREAD [Token]

Signature

$\rightarrow$	EXP ~ptr_void

5.1.2 Simple Locks

PRIM_MAKE_SIMPLE_LOCK [Token]

Signature

lock:	EXP pz
name:	EXP pz
$\rightarrow$	EXP pz

PRIM_DESTROY_SIMPLE_LOCK [Token]

Signature

lock_handle:	EXP pz
$\rightarrow$	EXP pz

PRIM_WAIT_FOR_SIMPLE_LOCK [Token]

Signature

lock:	EXP pz
$\rightarrow$	EXP pz

PRIM_WAIT_FOR_SIMPLE_LOCK_TIMED [Token]

Signature

lock:	EXP pz
timeout:	EXP pz
$\rightarrow$	EXP pz

PRIM_RELEASE_SIMPLE_LOCK [Token]

Signature

lock:	EXP pz
$\rightarrow$	EXP pz

PRIM_OWNED_SIMPLE_LOCK [Token]

Signature

lock_handle:	EXP pz
$\rightarrow$	EXP pz

5.1.3 Recursive Locks

PRIM_MAKE_RECURSIVE_LOCK [Token]

Signature

lock:	EXP pz
name:	EXP pz
$\rightarrow$	EXP pz

PRIM_DESTROY_RECURSIVE_LOCK [Token]

Signature

lock_handle:	EXP pz
$\rightarrow$	EXP pz

PRIM_WAIT_FOR_RECURSIVE_LOCK [Token]

Signature

lock:	EXP pz
$\rightarrow$	EXP pz

PRIM_WAIT_FOR_RECURSIVE_LOCK_TIMED [Token]

Signature

lock:	EXP pz
timeout:	EXP pz
$\rightarrow$	EXP pz

PRIM_RELEASE_RECURSIVE_LOCK [Token]

Signature

lock:	EXP pz
$\rightarrow$	EXP pz

PRIM_OWNED_RECURSIVE_LOCK [Token]

Signature

lock_handle:	EXP pz
$\rightarrow$	EXP pz

5.1.4 Semaphores

PRIM_MAKE_SEMAPHORE [Token]

Signature

lock:	EXP pz
name:	EXP pz
initial:	EXP pz
max:	EXP pz
$\rightarrow$	EXP pz

PRIM_DESTROY_SEMAPHORE [Token]

Signature

lock_handle:	EXP pz
$\rightarrow$	EXP pz

PRIM_WAIT_FOR_SEMAPHORE [Token]

Signature

lock:	EXP pz
$\rightarrow$	EXP pz

PRIM_WAIT_FOR_SEMAPHORE_TIMED [Token]

Signature

lock:	EXP pz
timeout:	EXP pz
$\rightarrow$	EXP pz

PRIM_RELEASE_SEMAPHORE [Token]

Signature

lock:	EXP pz
$\rightarrow$	EXP pz

5.1.5 Notifications

PRIM_MAKE_NOTIFICATION [Token]

Signature

notification:	EXP pz
name:	EXP pz
$\rightarrow$	EXP pz

PRIM_DESTROY_NOTIFICATION [Token]

Signature

notification_handle:	EXP pz
$\rightarrow$	EXP pz

PRIM_WAIT_FOR_NOTIFICATION [Token]

Signature

notification:	EXP pz
lock:	EXP pz
$\rightarrow$	EXP pz

PRIM_WAIT_FOR_NOTIFICATION_TIMED [Token]

Signature

notification:	EXP pz
lock:	EXP pz
timeout:	EXP pz
$\rightarrow$	EXP pz

PRIM_RELEASE_NOTIFICATION [Token]

Signature

notification:	EXP pz
lock:	EXP pz
→	EXP pz

PRIM_RELEASE_ALL_NOTIFICATION [Token]

Signature

notification:	EXP pz
lock:	EXP pz
→	EXP pz

5.1.6 Timers

PRIM_SLEEP [Token]

Signature

millisecs:	EXP pz
→	EXP pz

5.1.7 Thread Variables

PRIM_ALLOCATE_THREAD_VARIABLE [Token]

Signature

initial_value:	EXP pz
→	EXP ~ptr_void

5.2 ANDF Token Support for Atomic and Fluid Variables

Reading and writing of atomic and fluid variables requires special treatment in the compiler. Access to an atomic variable by one thread must be protected from accesses by other threads. Fluid variables require storage that is local to each thread, and access to a thread variable must obtain or update the value that is local to the calling thread.

The Dylan compiler front end handles atomic and fluid variables by defining two specialized classes of leaf nodes in the compiler's Implicit Continuation Representation (ICR): **<atomic-global-variable-leaf>** and **<fluid-global-variable-leaf>**. The ANDF back end for the compiler uses methods specialized on these classes to emit tokens that represent access to atomic variables (READ_ATOMIC, WRITE_ATOMIC, and CONDITIONAL_UPDATE_MEMORY) and access to thread variables (READ_THREAD and WRITE_THREAD).

5.2.1 Atomic Variable Access

READ_ATOMIC [Token]

Signature

var:	EXP ~ptr_void
$\rightarrow$	EXP pz

WRITE_ATOMIC [Token]

Signature

var:	EXP ~ptr_void
newval:	EXP pz
$\rightarrow$	EXP pz

CONDITIONAL_UPDATE_MEMORY [Token]

Signature

location:	EXP ~ptr_void
newval:	EXP pz
oldval:	EXP pz
$\rightarrow$	EXP pz

5.2.2 Thread Variables

READ_THREAD [Token]

Signature

variable_handle:	EXP ~ptr_void
$\rightarrow$	EXP pz

WRITE_THREAD [Token]

Signature

variable_handle:	EXP ~ptr_void
newval:	EXP pz
$\rightarrow$	EXP pz

6. Solaris Runtime Implementation

6.1 Expansion of ANDF Tokens Corresponding to Primitives

In the Solaris runtime,.there is a direct correspondence between ANDF tokens that represent primitives and C functions that implement Dylan thread support for Solaris. Each ANDF token expands into a call, using **apply_proc**, to the run-time routine that actually implements the functionality of the primitive defined by the portability interface.

6.1.1 Threads

PRIM_MAKE_THREAD [Token]

Expansion

(apply_proc pz (obtain_tag primitive_make_thread) thread_object name priority function)

PRIM_DESTROY_THREAD [Token]

Expansion

(apply_proc pz (obtain_tag primitive_destroy_thread) thread_handle)

PRIM_INITIALIZE_CURRENT_THREAD [Token]

Expansion

(apply_proc pz (obtain_tag primitive_initialize_current_thread) thread)

PRIM_THREAD_JOIN_SINGLE [Token]

Expansion

(apply_proc pz (obtain_tag primitive_thread_join_single) thread_handle)

PRIM_THREAD_JOIN_MULTIPLE [Token]

Expansion

(apply_proc pz (obtain_tag primitive_thread_join_multiple) thread_list)

PRIM_THREAD_YIELD [Token]

Expansion

(apply_proc pz (obtain_tag primitive_thread_yield))

PRIM_CURRENT_THREAD [Token]

Expansion

(apply_proc ~ptr_void (obtain_tag primitive_current_thread))

6.1.2 Simple Locks

PRIM_MAKE_SIMPLE_LOCK [Token]

Expansion

(apply_proc pz (obtain_tag primitive_make_simple_lock) lock name)

PRIM_DESTROY_SIMPLE_LOCK [Token]

Expansion

(apply_proc pz (obtain_tag primitive_destroy_simple_lock) lock_handle)

PRIM_WAIT_FOR_SIMPLE_LOCK [Token]

Expansion

(apply_proc pz (obtain_tag primitive_wait_for_simple_lock) lock)

PRIM_WAIT_FOR_SIMPLE_LOCK_TIMED [Token]

Expansion

(apply_proc pz (obtain_tag primitive_wait_for_simple_lock_timed) lock timeout)

PRIM_RELEASE_SIMPLE_LOCK [Token]

Expansion

(apply_proc pz (obtain_tag primitive_release_simple_lock) lock)

PRIM_OWNED_SIMPLE_LOCK [Token]

Expansion

(apply_proc pz (obtain_tag primitive_owned_simple_lock) lock_handle)

6.1.3 Recursive Locks

PRIM_MAKE_RECURSIVE_LOCK [Token]

Expansion

(apply_proc pz (obtain_tag primitive_make_recursive_lock) lock name)

PRIM_DESTROY_RECURSIVE_LOCK [Token]

Expansion

(apply_proc pz (obtain_tag primitive_destroy_recursive_lock) lock_handle)

PRIM_WAIT_FOR_RECURSIVE_LOCK [Token]

Expansion

(apply_proc pz (obtain_tag primitive_wait_for_recursive_lock) lock)

PRIM_WAIT_FOR_RECURSIVE_LOCK_TIMED [Token]

Expansion

(apply_proc pz (obtain_tag primitive_wait_for_recursive_lock_timed) lock timeout)

PRIM_RELEASE_RECURSIVE_LOCK [Token]

Expansion

(apply_proc pz (obtain_tag primitive_release_recursive_lock) lock)

PRIM_OWNED_RECURSIVE_LOCK [Token]

Expansion

(apply_proc pz (obtain_tag primitive_owned_recursive_lock) lock_handle)

6.1.4 Semaphores

PRIM_MAKE_SEMAPHORE [Token]

Expansion

(apply_proc pz (obtain_tag primitive_make_semaphore) lock name initial max)

PRIM_DESTROY_SEMAPHORE [Token]

Expansion

(apply_proc pz (obtain_tag primitive_destroy_semaphore) lock_handle)

PRIM_WAIT_FOR_SEMAPHORE [Token]

Expansion

(apply_proc pz (obtain_tag primitive_wait_for_semaphore) lock)

PRIM_WAIT_FOR_SEMAPHORE_TIMED [Token]

Expansion

(apply_proc pz (obtain_tag primitive_wait_for_semaphore_timed) lock timeout)

PRIM_RELEASE_SEMAPHORE [Token]

Expansion

(apply_proc pz (obtain_tag primitive_release_semaphore) lock)

6.1.5 Notifications

PRIM_MAKE_NOTIFICATION [Token]

Expansion

(apply_proc pz (obtain_tag primitive_make_notification) notification name)

PRIM_DESTROY_NOTIFICATION [Token]

Expansion

(apply_proc pz (obtain_tag primitive_destroy_notification) notification_handle)

PRIM_WAIT_FOR_NOTIFICATION [Token]

Expansion

(apply_proc pz (obtain_tag primitive_wait_for_notification) notification lock)

PRIM_WAIT_FOR_NOTIFICATION_TIMED [Token]

Expansion

**(apply_proc pz (obtain_tag primitive_wait_for_notification_timed)
notification lock timeout)**

PRIM_RELEASE_NOTIFICATION [Token]

Expansion

(apply_proc pz (obtain_tag primitive_release_notification) notification lock)

PRIM_RELEASE_ALL_NOTIFICATION [Token]

Expansion

(apply_proc pz (obtain_tag primitive_release_all_notification) notification)

6.1.6 Timers

PRIM_SLEEP [Token]

Expansion

(apply_proc pz (obtain_tag primitive_sleep) millisecs)

6.1.7 Thread Variables

PRIM_ALLOCATE_THREAD_VARIABLE [Token]

Expansion

(apply_proc ~ptr_void (obtain_tag primitive_allocate_thread_variable) initial_value)

6.2 Expansion of ANDF Tokens Supporting Atomic and Fluid Variables

Most uses of ANDF tokens that represent access to atomic variables and thread variables expand to calls to C functions that implement the associated operations. The exception is READ_ATOMIC, which represents reading an atomic variable. This token expands to a simple variable read, which in Solaris is guaranteed to be atomic. This section lists expansions of the remaining tokens that support atomic and thread variables.

6.2.1 Atomic Variable Access

WRITE_ATOMIC [Token]

Expansion

(apply_proc pz (obtain_tag primitive_assign_atomic_memory) location newval)

CONDITIONAL_UPDATE_MEMORY [Token]

Expansion

(apply_proc pz (obtain_tag primitive_conditional_update_memory) location newval oldval)

6.2.2 Thread Variables

READ_THREAD [Token]

Expansion

(apply_proc pz (obtain_tag primitive_read_thread_variable) variable_handle)

WRITE_THREAD [Token]

Expansion

(apply_proc pz (obtain_tag primitive_write_thread_variable) variable_handle, newval)

6.3 Solaris Runtime Primitive Reference

6.3.1 Threads

primitive_make_thread [Function]

Signature

void primitive_make_thread (D_THREAD* thread, D_NAME name, ZINT priority, ZFN function);

Implementation

The primitive starts a new Solaris thread, and causes it to call the supplied Dylan function. To do this, it first stores the Dylan function in one of the **D_THREAD** container slots (because Solaris does not allow for passing multiple parameters at thread creation time). It then calls the Solaris function **thr_create** passing it, among other arguments, a C function to call named **trampoline,** and the **D_THREAD** Dylan thread object. **thr_create** starts the new thread which calls **trampoline** with the thread object as an argument.

The **trampoline** function first retrieves the function object in the container slot of the **D_THREAD** object. It then initialises the data structures used to implement **join-thread** and the thread-local variables, and it modifies the container slots of the thread object. It then uses Dylan conventions to call the function object with no arguments. When the Dylan function returns, the thread is potentially ready for joining, so **trampoline** updates some data structures and exits.

Dylan threads are implemented as *unbound* threads (i.e., they are not bound to a specific lightweight process). This means that the operating system does not guarantee to honour the priority levels for a thread, even though they are set by the primitive.

The **name** and **function** arguments are ignored.

primitive_destroy_thread [Function]

Signature

void primitive_destroy_thread (D_THREAD* thread);

Implementation

This function is designed to free any runtime-allocated memory associated with the creation of the Solaris thread object. However, Solaris thread handles are represented as integers and there is no allocated memory to free. The function thus does nothing.

primitive_initialize_current_thread [Function]

Signature

void primitive_initialize_current_thread (D_THREAD* thread);

Implementation

This function is used to initialize the first thread, which is not created by the Dylan runtime system. The container slots in the Dylan thread object are destructively modified with the handles of the current Solaris thread, and the data structures used to implement **join-thread** and the thread-local variables are initialized. This initialization is equivalent to that performed in the **trampoline** function, for threads created with **primitive_make_thread**.

primitive_thread_join_single [Function]

Signature

ZINT primitive_thread_join_single (D_THREAD* thread);

Implementation

This function blocks until the supplied Dylan thread object has finished execution. The semantics are similar to the Solaris function **thr_join**, except that joining of threads which have already exited is permitted. The **handle2** field of the **D_THREAD** thread object is used to maintain the state information necessary to support this, and contains one of three states:

1. **Normal**: The thread is still running. Set by **trampoline** (via **primitive_make_thread**) before calling the Dylan function.
2. **Marked**: The thread has finished, but no join has yet been recorded. Set by **trampoline** after the Dylan function has returned.
3. **Joined**: At the Dylan level, a join has already occurred. Set by **primitive_thread_join_single and primitive_thread_join_multiple.** Requesting another join on the same thread is a Dylan error.

The primitive first checks the state of the specified Dylan thread, to see whether it has finished. The testing and updating of the state is carried out under the protection of a single mutual exclusion lock. If the state is **Normal**, the lock is released and the **thr_join** function is called to block until the Dylan thread finishes; the state is then checked again. If the state is **Marked**, the state is updated to **Joined**, the lock is released, and the primitive returns to indicate a successful join. If the state is **Joined**, the lock is released, and an error value is returned.

primitive_thread_join_multiple [Function]

Signature

Z primitive_thread_join_multiple (SOV* thread_vector);

Implementation

This primitive checks the state of all threads in its **SOV** vector in similar fashion to **primitive_thread_join_single**. The primitive loops continually, first testing the state of all the thread objects in **thread_vector** to see if any has finished, and then sleeping. The testing phase, and any state updating, is performed under the protection of the same lock as used by **primitive_thread_join_single**.

During the testing phase, if a **Marked** thread is found, then its state is set to **Joined**, the lock is released, and the primitive returns to indicate a successful join. If a **Joine**d thread is found the lock is released, and an error value is returned.

If no **Marked** or **Joined** threads are found, then the primitive releases the lock, and then enters a sleeping phase, by calling **primitive_sleep** with an argument of 10 milliseconds. Finally it repeats the loop, in case the state of any of the threads has changed.

primitive_thread_yield [Function]

Signature

void primitive_thread_yield (void);

Implementation

The Solaris **thr_yield** function is used directly.

primitive_current_thread [Function]

Signature

void* primitive_current_thread (void);

Implementation

The Solaris **thr_self** function is used directly to return the Solaris identifier of the current thread. This corresponds with the contents of the **handle1** field of one of the Dylan thread objects, known to the Dylan library.

6.3.2 Simple Locks

primitive_make_simple_lock [Function]

Signature

void primitive_make_simple_lock (CONTAINER* lock, D_NAME name);

Implementation

The lock object is destructively updated so that its handle slot refers to a handle on a Solaris provided mutex. A call to **malloc** is made for the lock, and the Solaris **mutex_init** function is used directly to initialize the mutex. The Solaris mutex is stored in the **handle** field of the Dylan lock.

primitive_destroy_simple_lock [Function]

Signature

void primitive_destroy_simple_lock (CONTAINER* lock);

Implementation

The Solaris **mutex_destroy** function is used directly, and the relevant memory freed.

primitive_wait_for_simple_lock [Function]

Signature

ZINT primitive_wait_for_simple_lock (CONTAINER* lock);

Implementation

The Solaris **mutex_lock** function is used.

primitive_wait_for_simple_lock_timed [Function]

Signature

ZINT primitive_wait_for_simple_lock_timed (CONTAINER* lock, ZINT millisecs);

Implementation

The timeout period (in milliseconds) is a given as a tagged integer which has to be converted into a normal C integer before use. This is in turn converted into an absolute time by manipulating the results of calls to the Solaris **clock_gettime** function. The results are returned to the calling primitive as a Solaris-specific data type.

The primitive then enters a **while** loop which checks that the absolute timeout time has not been passed, and then tries to acquire the relevant lock using the Solaris **mutex_trylock** function, which does not block on failure. If the lock is acquired, then 0 is returned to indicate success. If not, **primitive_sleep** is called with an argument of 10 milliseconds, after which the loop continues. If the call ultimately times out, an integer indicating failure to acquire the lock is returned.

primitive_release_simple_lock [Function]

Signature

ZINT primitive_release_simple_lock (CONTAINER* lock);

Implementation

The Solaris **mutex_unlock** function is used.

primitive_owned_simple_lock [Function]

Signature

ZINT primitive_owned_simple_lock (CONTAINER* lock);

Implementation

Information on the ownership of the lock is extracted from the Solaris structures and used to check that the lock is held by the calling thread.

6.3.3 Recursive Locks

primitive_make_recursive_lock [Function]

Signature

void primitive_make_recursive_lock (CONTAINER* lock, D_NAME name);

Implementation

Each recursive lock requires a dedicated structure with fields including a Solaris mutex lock, the identity of the thread holding it, and the current count for the number of times it has been locked/unlocked.

Calls to **malloc** are made for the lock and its associated structure. The Solaris function **mutex_init** is then used, and the recursive lock structure members are initialised. The newly allocated structure is stored in the **handle** field of the Dylan lock.

primitive_destroy_recursive_lock [Function]

Signature

void primitive_destroy_recursive_lock (CONTAINER* lock);

Implementation

The Solaris **mutex_destroy** function is used directly, and the relevant memory freed.

primitive_wait_for_recursive_lock [Function]

Signature

ZINT primitive_wait_for_recursive_lock (CONTAINER* lock);

Implementation

If the calling thread already holds the specified lock, then its count is increased by one. If not, the Solaris **mutex_lock** function is called to acquire it, and the lock's structure is updated to record the current owner and a count of 1, Finally, an integer is returned to indicate a successful result.

primitive_wait_for_recursive_lock_timed [Function]

Signature

ZINT primitive_wait_for_recursive_lock_timed (CONTAINER* lock, ZINT millisecs);

Implementation

If the calling thread already holds the specified lock, then its count is increased by one. Otherwise, the implementation is as for **primitive_wait_for_simple_lock_timed**, followed by an update of the owner and count fields.

primitive_release_recursive_lock [Function]

Signature

ZINT primitive_release_recursive_lock (CONTAINER* lock);

Implementation

The lock is first checked to ensure that the current thread is the owner, and an error value is returned if not. Otherwise, the lock's count is decremented. If the count is then zero, the Solaris **mutex_unlock** function is called.

primitive_owned_recursive_lock [Function]

Signature

ZINT primitive_owned_recursive_lock (CONTAINER* lock);

Implementation

The lock's associated structure is checked to see if the current thread is the owner.

6.3.4 Semaphores

primitive_make_semaphore [Function]

Signature

void primitive_make_semaphore (CONTAINER* lock, D_NAME name, ZINT initial, ZINT max);

Implementation

A call to **malloc** is made for the semaphore, and the Solaris **sema_init** function is used to initialize it. The Solaris semaphore is stored in the **handle** field of the Dylan lock.

primitive_destroy_semaphore [Function]

Signature

void primitive_destroy_semaphore (CONTAINER* lock);

Implementation

The Solaris **sema_destroy** function is used, and the relevant memory freed.

primitive_wait_for_semaphore [Function]

Signature

ZINT primitive_wait_for_semaphore (CONTAINER* lock);

Implementation

The Solaris **sema_wait** function is used.

primitive_wait_for_semaphore_timed [Function]

Signature

ZINT primitive_wait_for_semaphore_timed (CONTAINER* lock, ZINT millisecs);

Implementation

This primitive uses the same timing mechanisms as **primitive_wait_for_simple_lock_timed**, except the loop tests for success with the Solaris function **sema_trywait,** which does not block on failure.

primitive_release_semaphore [Function]

Signature

ZINT primitive_release_semaphore (CONTAINER* lock);

Implementation

The Solaris **sema_post** function is used.

6.3.5 Notifications

primitive_make_notification [Function]

Signature

void primitive_make_notification (CONTAINER* notification, D_NAME name);

Implementation

A call to **malloc** is made for a condition variable, and the Solaris **cond_init** function is used directly. The Solaris condition variable is stored in the **handle** field of the Dylan lock.

primitive_destroy_notification [Function]

Signature

void primitive_destroy_notification (CONTAINER* notification);

Implementation

The Solaris **cond_destroy** function is used directly, and the relevant memory freed.

primitive_wait_for_notification [Function]

Signature

ZINT primitive_wait_for_notification (CONTAINER* notification, CONTAINER* lock);

Implementation

Ownership of the lock is first tested with **primitive_owned_recursive_lock**. The Solaris **cond_wait** function is then used directly to wait for the signalling of a change in the Solaris condition variable.

primitive_wait_for_notification_timed [Function]

Signature

**ZINT primitive_wait_for_notification_timed (CONTAINER* notification,
CONTAINER* lock, ZINT millisecs);**

Implementation

Ownership of the lock is first tested with **primitive_owned_recursive_lock**. A test is then made to determine whether the specified timeout period (in milliseconds) is 0, less than 1000, or more than 1000. The Solaris **cond_timedwait** function is then used directly with an appropriately scaled timeout, depending on the range.

primitive_release_notification [Function]

Signature

ZINT primitive_release_notification (CONTAINER* notification, CONTAINER* lock);

Implementation

Ownership of the lock is first tested with **primitive_owned_recursive_lock**. The Solaris **cond_signal** function is then used directly.

primitive_release_all_notification [Function]

Signature

ZINT primitive_release_all_notification (CONTAINER* notification, CONTAINER* lock);

Implementation

Ownership of the lock is first tested with **primitive_owned_recursive_lock**. The Solaris **cond_broadcast** function is then used directly.

6.3.6 Timers

primitive_sleep [Function]

Signature

void primitive_sleep (ZINT millisecs);

Implementation

This version of Solaris does not provide a ready-made sleep primitive, so the conditional variable functions are used instead.

The **ZINT** value is converted from a tagged to a standard integer, and an absolute timeout value calculated. A dedicated mutex lock and condition variable are generated and initialised. This lock is acquired, and a call is made to the Solaris function **cond_timedwait**, giving a pointer to the absolute timeout time. When this call returns, the lock is released and the primitive itself returns.

6.3.7 Atomic Variable Access

primitive_assign_atomic_memory [Function]

Signature

Z primitive_assign_atomic_memory (void* location, Z newval);

Implementation

A single mutual exclusion lock **atomic_lock** is used to synchronise the updating of the specified location.

The function is implemented roughly as follows:-

```
mutex_lock(&atomic_lock);
*location = newval;
mutex_unlock(&atomic_lock);
return(newval);
```

primitive_conditional_update_memory [Function]

Signature

ZINT primitive_conditional_update_memory (void* location, Z newval, Z oldval);

Implementation

A single mutual exclusion lock **atomic_lock** is used to synchronise the updating of the specified location.

The function is implemented roughly as follows:

```
int res = 0;
mutex_lock(&atomic_lock);
if (*location == oldval)
        { *location = newval;
        res = 1;
        }
mutex_unlock(&atomic_lock);
return(res);
```

6.3.8 Thread Variables

primitive_allocate_thread_variable [Function]

Signature

void* primitive_allocate_thread_variable (Z initial_value);

Implementation

Thread variable allocations (corresponding to definitions of fluid variables) can be interleaved with thread creation. Also, the number of thread variables is not statically determinable. Whenever a thread variable is allocated, its initial value must be given to all threads which are in existence. Whenever a new thread is created it must assume all the default initial values known so far.

We use a single Solaris thread-local key to store a vector of all the Dylan thread-local variables. The Solaris runtime system will maintain indexes into this vector and will return these to Dylan as the thread-local variable handles.

A vector is maintained of the default initial values, and a private vector is used for every thread.

On thread creation, the **trampoline** function performs the following initialization steps:

1. Allocate a new vector of thread-local storage for the new thread.
2. Initialise it with a copy of the values in the default vector.
3. Under control of the lock t**hread_vector_lock**, insert this new vector into a linked list of all the currently active thread vectors.

Once the Dylan function has returned, the following cleanup operations are performed:

1. Under the control of **thread_vector_lock**, remove the vector from the list.
2. Free the vector.

On allocation of a new thread variable:

1. Increment the current index into the vector (global variable **max_key**).
2. Initialise the default vector at index **max_key** with the initial value.
3. Under control of **thread_vector_lock**, iterate over all the thread vectors in the list, and initialise the element at index **max_key** with the initial value.
4. Return the index into the vector as a handle on the thread variable.

For now, we assume that we do not overflow the variable vector. We allocate each vector as a suitably large size (e.g., 1024), and assume that we won't ever have that many fluid variables.

In the future, we can expand the variable vectors by using an extra indirection, and by replacing each vector with a new, bigger one, when overflow is detected. (This must be done with all other threads suspended).

A possible improvement to this idea is to use a two tier system, with the first *N* fluid variables implemented as described above, but with a second thread-local vector (and key, and extra indirection) for all other variables. The second vector can then be expanded on demand, but there is less of an overhead for the first *N* variables, aside from a test for whether the variable count is in the range 0–*N*.

primitive_read_thread_variable [Function]

Signature

Z primitive_read_thread_variable (void* handle_on_variable);

Implementation

The argument is assumed to be an integer index into the vector of thread-local variables, as returned by **primitive_allocate_thread_variable**. This primitive uses the Solaris **thr_getspecific** function to retrieve the variable vector that contains values local to the calling thread. It returns the value at the specified index into the variable vector.

primitive_write_thread_variable [Function]

Signature

Z primitive_write_thread_variable (void* handle_on_variable, Z newval);

Implementation

The first argument is assumed to be an integer index into the vector of thread-local variables, as returned by **primitive_allocate_thread_variable**. This primitive uses the Solaris **thr_getspecific** function to retrieve the variable vector that contains values local to the calling thread. It stores the new value at the specified index into the variable vector and returns the new value.

7. Performance Evaluation

The intention of the Simple Dylan Threads Library is to provide threads facilities via operating system API calls. ANDF helps to achieve this goal by providing an abstract interface layer of tokens. In most cases, the preferred implementation of these tokens will be to expand directly into calls into the Dylan runtime system — in which case ANDF has no significant effect on performance. This will be true for all of the ANDF tokens which correspond to primitive functions to the Dylan compiler, at least for all general purpose target platforms where threads support is provided by the operating system.

For those tokens corresponding to the access of atomic variables and fluid variables, there are various possibilities for token expansion depending on the target platform. On some platforms ANDF may impose a cost for the usage of these variables.

It is assumed that atomic variables can always be represented as normal global variables, and that it should always be possible to read the value of the variable in the normal way. The implementation choice for conditional update of an atomic variable, however, will depend on whether the platform has hardware support for this operation (such as Compare And Exchange on the Intel 486, or Store Conditional on the DEC Alpha). If no hardware support is available, then assignments and conditional updates to the variable must be protected by a lock. ANDF can describe the operations involved in an equivalent manner to any native implementation, and will impose no extra cost in this case. However, if hardware support is available, it is highly desirable that it is used. Ideally the installer for the target will be able to generate the appropriate instructions by means of a platform-specific token. In this case, again, there will be no extra cost for using ANDF. But in cases where the platform provides hardware support but the installer cannot generate the instructions, there will indeed be a cost for using ANDF on that platform. At the worst, this cost will be the cost of a function call, since the ANDF token can always expand into a call into the runtime system, which can provide an appropriate usage of the instruction.

As with atomic variables, thread-local variables may or may not be most efficiently implemented with special hardware instructions (such as a load from memory referenced by a special segment descriptor register for the Intel 486). Operating systems which provide threads typically provide some API to access thread-local variables, and ANDF will be able to access this with no extra cost if this is the preferred option for that operating system. If there is a more efficient hardware instruction, however, then the performance impact of using ANDF on that platform again depends on whether the installer is able to generate the hardware instruction. As for atomic variables, this cost is limited to a single function call.

8. Conclusions

We have shown that it is possible to implement Dylan, an advanced dynamic language with support for garbage collection, multiple threads and interoperability with other languages, using ANDF as a portable medium. This is achieved through the usage of platform specific token expansions, and a platform specific runtime system.

The complexities of implementing the dynamic features of Dylan along with the threads support are minimized, because the runtime system can be written for each platform with explicit knowledge of the implementation of all of the features and how they interact. However, since these implementation details are not described in the portable ANDF code, there may be a significant cost for porting the implementation of Dylan to a new platform. This is a once only cost, however. All ANDF produced from portable Dylan programs should be runnable on the new platform once the platform-specific tokens and runtime system have been implemented.

The parallelism features of the threads library include threads themselves and synchronization mechanisms. These are expected to be implemented as API calls to operating system threads facilities, since this approach has the advantage that Dylan components can then inter-operate with other components, even if multiple threads are being used. Most tokens will probably expand into runtime system API calls, and the runtime system will then call the operating system indirectly. This design permits any mapping between the threads library requirements and the operating systems facilities to be made out of line, which reduces code size. However, ANDF does not constrain such an implementation, either in terms of performance or implementation choice, since ANDF token expansions may be used to describe appropriate API calls for each platform.

There are some features in the Dylan threads library which might depend upon hardware support for ultimate efficiency, on platforms where this is available. These include conditional update of variables, and thread-local variables. There may be an implementation cost for these features if the ANDF installer cannot generate the appropriate hardware instructions directly.

9. References and Bibliography

[Apple94] Apple Computer, “Dylan (TM) Interim Reference Manual”, Browsable on the World Wide Web through http://www.cambridge.apple.com/dylan/dylan.html, April 1994

[BW88] Boehm, H-J., Weiser, M., “Garbage Collection in an Uncooperative Environment”, Software Practice and Experience, vol. 18, #9, September 1988.

[BDS91] Boehm, H-J., Demers, A.J., Shenker, S., “Mostly Parallel Garbage Collection”, Proceedings of SIGPLAN ‘91 Conference on Programming Language Design and Implementation, June 1991, ACM

[Edwards93]Edwards, P., “Preliminary Parallelism Extensions”, GLUE Deliverable 5.1.1, DRA Malvern, October 1993

[Mann93a] Mann, T., “Initial evaluation of TDF Support for Garbage Collection”, GLUE Deliverable 4.2.2a, Harlequin Ltd., 1993.

[ME94] Mann, T., Miranda, E., “Requirements Specification for Parallel Extensions to TDF to Support Dylan — Including Run-Time Support”, GLUE Deliverable 5.7.1, Harlequin Ltd., 1994.

[MG95] Mann, T., Green, S., “Report on Inter-Language Working in ANDF”, GLUE Deliverable 4.2.3, Harlequin Ltd., 1995.

[Sun93] Sun Microsystems, Inc., “Guide to Multithread Programming”, SunOS 5.3, November 1993.

[TDF92] Foster, M., Currie, I., “TDF Specification”, DRA Malvern, September 1992.