Cleavir

Cutting edge compilation tools for Common Lisp

Robert Strandh

2014

Contents

Chapter 1

Introduction

Cleavir is an implementation-independent framework for creating Common Lisp compilers.

It is implementation-independent in that it provides:

- general features that every implementation needs,
- features that implementations can optionally choose to take advantage of,
- alternative features that are appropriate for some implementations and not for others,
- mechanisms for allowing implementation-specific features that integrate seamlessly into the general framework.

To use the framework, an implementation provides methods for a few generic functions that allow Cleavir to access the compilation and evaluation environments of the implementation. Cleavir uses those functions to turn a form into an *abstract syntax tree*, or AST. (See Chapter 4.) Cleavir then translates the AST into an intermediate representation called HIR.[1] (See Chapter 6.) This

[1] HIR stands for High-level Intermediate Representation.

translation can be customized by the implementation. Cleavir then provides a number of transformations on this representation such as:

- translation into *static single assignment* form,
- type inference,
- standard optimizations such as value numbering, common subexpression elimination, redundancy elimination, etc.

Once these implementation-independent and backend-independent transformations have been accomplished, the HIR notation is gradually transformed into a notation that is specific both to the implementation and to the backend. The result of the first step of this transformation is an intermediate form called MIR.[2] MIR differs from HIR in that it exposes address calculations and memory accesses using those addresses, whereas HIR manipulates only Common Lisp objects. The notation becomes even more backend-specific in that it introduces features such as registers and stack frames. The notation also becomes implementation specific in that it exposes choices such as argument passing.

Finally, the low-level code is translated into machine code.

At each stage, an implementation can customize the process by introducing new classes and methods.

[2] MIR means Medium-level Intermediate Representation.

Chapter 2

Environment

2.1 Introduction

Translating a Common Lisp program from source code into an abstract syntax tree is done in constant interaction with an *environment*. The HyperSpec stipulates[1] that there are four different environments that are relevant to compilation:

- The *startup environment*. This environment is the global environment of the Common Lisp system when the compiler was invoked.
- The *compilation environment*. This environment is the local environment in which forms are compiled. It is also the environment that is passed to macro expanders.
- The *evaluation environment*. According to the HyperSpec, this environment is "a run-time environment in which macro expanders and code specified by `eval-when` to be evaluated are evaluated".
- The *run-time environment*. This environment is used when the resulting compiled program is executed.

[1] See Section 3.2.1 of the HyperSpec.

The HyperSpec does not specify how environments are represented, and there is no specified protocol for manipulating environments. As a result, each implementation has its own representation and its own protocols.

In this chapter, we define a protocol for accessing the compilation environment during compilation. When Cleavir is asked to convert a form to an abstract syntax tree, client code must supply an object that represents the startup environment. During the conversion process, Cleavir will call the functions documented in Section 2.4 to augment the startup environment with information introduced by binding forms to form an augmented compilation environment. To determine the meaning of the program elements in the form to be converted, Cleavir will call the functions documented in Section 2.3.

Client code must supply methods on the functions in Section 2.4 to augment the environment and return the resulting environment. Client code must also supply methods on the functions in Section 2.3 that will query the environment (whether the startup environment or the augmented environment) and return the relevant information to Cleavir.

It might seem that Cleavir could represent the *local part* of the environment (i.e., the part of the environment that is temporarily introduced when nested forms are compiled) in whatever way it pleases, but this is not the case. The reason is that the full environment must be passed as an argument to macro expanders that are defined in the startup environment, and those macro expanders are implementation specific. It is also not possible for Cleavir to define its own version of `macroexpand`, because a globally defined implementation-specific macro expander may call the implementation-specific version of `macroexpand` which would fail if given an environment other than the one defined by the implementation.

Cleavir does, however, provide some features that are useful for an incomplete implementation for which no representation of the local part of the environment has been determined. The creator of such an implementation may choose to use the *default representation* of that local part of the environment supplied by Cleavir. The default version can also be used for a more complete implementations, provided that the implementation-specific version of `macroexpand` is not called on an environment using this default representation.

2.2 Package

All symbols documented in this chapter have as their home package the package named `cleavir-environment`. This package has a single nickname which is `cleavir-env`. We strongly advise against `:use`-ing this package, because it contains several symbols that have the same name as symbols in the package named `common-lisp`. Instead, we advise that client code use symbols in this package with their explicit package prefix.

2.3 Querying the environment

In this section, we describe classes and functions that are used by the compiler to query the environment concerning information about program elements that the compiler needs in order to determine how to process those program elements.

When the compiler calls a generic query function, it passes the environment as the first argument. Client code must supply methods on these functions, specialized to its particular representation of environments.

These methods should return instances of the classes described in this section. Any such instance contains all available information about some program element in that particular environment. This information must typically be assembled from different parts of the environment. For that reason, client code typically creates a new instance whenever a query function is called, rather than attempting to store such instances in the environment. If any of these client-supplied methods fails to accomplish its task, it should return `nil`.

Client code is free to define subclasses of the classes described here, for instance in order to represent implementation-specific information about the program elements. Client code would then typically also provide auxiliary methods or overriding primary methods on the compilation functions that handle these classes.

2.3.1 Variable information

⇒ `variable-info` *environment symbol* [*Generic Function*]

This function is called by the compiler whenever a symbol in a *variable* position is to be compiled. It returns an instance of one of the classes described below.

⇒ `no-variable-info` [*Condition*]

This condition is signaled by Cleavir when a client-supplied method on the generic function `variable-info` returns `nil`.

⇒ `name` *(condition* `no-variable-info`*)* [*Method*]

This method returns the name of the variable for which no info was available.

Lexical variable information

⇒ `lexical-variable-info` [*Class*]

This class represents information about lexical variables. An instance of this class is returned by a call to `variable-info` when it turns out that the symbol passed as an argument refers to a lexical variable.

⇒ `:name` [*Initarg*]

This initarg supplies the name of the lexical variable. This initarg must be supplied.

⇒ `:identity` [*Initarg*]

This initarg is used to supply some kind of implementation-defined *identity*. The implementation can supply any object as the identity, because it is not interpreted by the compiler. However, the *same* identity must be supplied each time for a particular lexical variable. This initarg must be supplied.

⇒ `:type` [*Initarg*]

This initarg is used to supply the *type* of the lexical variable. The type can be any type specifier and it may contain user-defined types. If this initarg is omitted, it defaults to `t`.

⇒ `:ignore` [*Initarg*]

This initarg is used to supply *ignore* information about the lexical variable. The value of this initarg can be either `ignore` (i.e., the symbol with that name in the `common-lisp` package) meaning that the variable is declared `ignore`, `ignorable` (i.e., the symbol with that name in the `common-lisp` package) meaning that the variable is declared `ignorable`, or `nil` meaning that no ignore or ignorable declaration for this variable is in scope. If this initarg is not supplied, it defaults to `nil`.

⇒ `:dynamic-extent` [*Initarg*]

This initarg is used to supply *dynamic extent* information about the lexical variable. The value of this initarg can be either `t`, meaning that the variable has been declared `dynamic-extent`, or `nil`, meaning that the variable has not been declared `dynamic-extent`. The default value when this initarg is not supplied is `nil`.

⇒ `name` *(info* `lexical-variable-info`*)* [*Method*]

Given an instance of the class `lexical-variable-info`, this method returns the name of the lexical variable as supplied by the initarg `:name`.

⇒ `identity` *(info* `lexical-variable-info`*)* [*Method*]

Given an instance of the class `lexical-variable-info`, this method returns the identity of the lexical variable as supplied by the initarg `:identity`.

⇒ `type` *(info* `lexical-variable-info`*)* [*Method*]

Given an instance of the class `lexical-variable-info`, this method returns the *type* of the variable as supplied by the initarg `:type`. If that initarg was not supplied, this method returns `t`.

⇒ `ignore` *(info* `lexical-variable-info`*)* [*Method*]

Given an instance of the class `lexical-variable-info`, this method returns the *ignore* information of the lexical variable as supplied by the initarg `:ignore`. If that initarg was not supplied, this method returns `nil`.

⇒ `dynamic-extent` *(info* `lexical-variable-info`*)* [*Method*]

Given an instance of the class `lexical-variable-info`, this method returns the *dynamic extent* information of the lexical variable as supplied by the initarg `:dynamic-extent`. If that initarg was not supplied, this method returns `nil`.

Special variable information

⇒ `special-variable-info` [*Class*]

This class represents information about special variables. An instance of this class is returned by a call to `variable-info` when it turns out that the symbol passed as an argument refers to a special variable.

⇒ `:name` [*Initarg*]

This initarg supplies the name of the special variable. This initarg must be supplied.

⇒ `:type` [*Initarg*]

This initarg is used to supply the *type* of the special variable. The type can be any type specifier and it may contain user-defined types. If this initarg is omitted, it defaults to `t`.

⇒ `:ignore` [*Initarg*]

This initarg is used to supply *ignore* information about the special variable. The value of this initarg can be either `ignore` (i.e., the symbol with that name in the `common-lisp` package) meaning that the variable is declared `ignore`, `ignorable` (i.e., the symbol with that name in the `common-lisp` package) meaning that the variable is declared `ignorable`, or `nil` meaning that no ignore or ignorable declaration for this variable is in scope. If this initarg is not supplied, it defaults to `nil`.

⇒ `name` *(info* `special-variable-info`*)* [*Method*]

Given an instance of the class `special-variable-info`, this method returns the name of the special variable as supplied by the initarg `:name`.

⇒ `type` *(info* `special-variable-info`*)* [*Method*]

Given an instance of the class `special-variable-info`, this method returns the *type* of the variable as supplied by the initarg `:type`. If that initarg was not supplied, this method returns `t`.

⇒ `ignore` *(info* `special-variable-info`*)* [*Method*]

Given an instance of the class `special-variable-info`, this method returns

the *ignore* information of the special variable as supplied by the initarg `:ignore`. If that initarg was not supplied, this method returns `nil`.

Constant variable information

⇒ `constant-variable-info` [*Class*]

This class represents information about constant variables. An instance of this class is returned by a call to `variable-info` when it turns out that the symbol passed as an argument refers to a constant variable.

⇒ `:name` [*Initarg*]

This initarg supplies the name of the constant variable. This initarg must be supplied.

⇒ `:value` [*Initarg*]

This initarg supplies the value of the constant variable. This initarg must be supplied.

⇒ `name` *(info* `constant-variable-info`*)* [*Method*]

Given an instance of the class `constant-variable-info`, this method returns the name of the constant variable as supplied by the initarg `:name`.

⇒ `value` *(info* `constant-variable-info`*)* [*Method*]

Given an instance of the class `constant-variable-info`, this method returns the value of the constant variable as supplied by the initarg `:value`.

Symbol macro information

⇒ `symbol-macro-info` [*Class*]

This class represents information about symbol macros. An instance of this class is returned by a call to `variable-info` when it turns out that the symbol passed as an argument refers to a symbol macro.

⇒ `:name` [*Initarg*]

This initarg supplies the name of the symbol macro. This initarg must be

supplied.

⇒ `:expansion` [*Initarg*]

This initarg supplies the expansion of the symbol macro. This initarg must be supplied.

⇒ `:type` [*Initarg*]

This initarg is used to supply the *type* of the symbol macro. The type can be any type specifier and it may contain user-defined types. If this initarg is omitted, it defaults to `t`.

⇒ `name` *(info* `symbol-macro-info`*)* [*Method*]

Given an instance of the class `symbol-macro-info`, this method returns the name of the symbol macro as supplied by the initarg `:name`.

⇒ `expansion` *(info* `symbol-macro-info`*)* [*Method*]

Given an instance of the class `symbol-macro-info`, this method returns the expansion of the symbol macro as supplied by the initarg `:expansion`.

⇒ `type` *(info* `symbol-macro-info`*)* [*Method*]

Given an instance of the class `symbol-macro-info`, this method returns the *type* of the symbol macro as supplied by the initarg `:type`. If that initarg was not supplied, this method returns `t`.

2.3.2 Function information

⇒ `function-info` *environment function-name* [*Generic Function*]

This function is called by the compiler whenever a symbol in a *function* position is to be compiled or whenever a function name is found in a context where it is known to refer to a function. It returns an instance of one of the classes described below.

⇒ `no-function-info` [*Condition*]

This condition is signaled by Cleavir when a client-supplied method on the generic function `function-info` returns `nil`.

⇒ `name` *(condition* `no-function-info`*)* [*Method*]

This method returns the name of the function for which no info was available.

Local function information

⇒ `local-function-info` [*Class*]

This class represents information about local functions introduced by `flet` or `labels`. An instance of this class is returned by a call to `function-info` when it turns out that the function name passed as an argument refers to a local function.

⇒ `:name` [*Initarg*]

This initarg supplies the name of the local function. This initarg must be supplied.

⇒ `:identity` [*Initarg*]

This initarg is used to supply some kind of implementation-defined *identity*. The implementation can supply any object as the identity, because it is not interpreted by the compiler. However, the *same* identity must be supplied each time for a particular local function. This initarg must be supplied.

⇒ `:type` [*Initarg*]

This initarg is used to supply the *type* of the local function. The type can be any function type specifier and it may contain user-defined types. If this initarg is omitted, it defaults to `t`.

⇒ `:inline` [*Initarg*]

This initarg is used to supply *inline* information about the local function. The value of this initarg can be either `inline` (i.e., the symbol with that name in the `common-lisp` package) meaning that the function is declared `inline`, `notinline` (i.e., the symbol with that name in the `common-lisp` package) meaning that the function is declared `notinline`, or `nil` meaning that no inline declaration for this function is in scope. If this initarg is not supplied, it defaults to `nil`.

⇒ `:ignore` [*Initarg*]

This initarg is used to supply *ignore* information about the local function. The value of this initarg can be either `ignore` (i.e., the symbol with that name in the `common-lisp` package) meaning that the function is declared `ignore`, `ignorable` (i.e., the symbol with that name in the `common-lisp` package) meaning that the function is declared `ignorable`, or `nil` meaning that no ignore or ignorable declaration for this function is in scope. If this initarg is not supplied, it defaults to `nil`.

⇒ `:dynamic-extent` [*Initarg*]

This initarg is used to supply *dynamic extent* information about the local function. The value of this initarg can be either `t`, meaning that the function has been declared `dynamic-extent`, or `nil`, meaning that the function has not been declared `dynamic-extent`. The default value when this initarg is not supplied is `nil`.

⇒ `name` *(info* `local-function-info`*)* [*Method*]

Given an instance of the class `local-function-info`, this method returns the name of the local function as supplied by the initarg `:name`.

⇒ `identity` *(info* `local-function-info`*)* [*Method*]

Given an instance of the class `local-function-info`, this method returns the identity of the local function as supplied by the initarg `:identity`.

⇒ `type` *(info* `local-function-info`*)* [*Method*]

Given an instance of the class `local-function-info`, this method returns the *type* of the local function as supplied by the initarg `:type`. If that initarg was not supplied, this method returns `t`.

⇒ `inline` *(info* `local-function-info`*)* [*Method*]

Given an instance of the class `local-function-info`, this method returns the *inline* information of the local function as supplied by the initarg `:inline`. If that initarg was not supplied, this method returns `nil`.

⇒ `ignore` *(info* `local-function-info`*)* [*Method*]

Given an instance of the class `local-function-info`, this method returns the *ignore* information of the local function as supplied by the initarg `:ignore`. If that initarg was not supplied, this method returns `nil`.

⇒ `dynamic-extent` *(info* `local-function-info`*)* [*Method*]

Given an instance of the class `local-function-info`, this method returns the *dynamic extent* information of the local function as supplied by the initarg `:dynamic-extent`. If that initarg was not supplied, this method returns `nil`.

Global function information

⇒ `global-function-info` [*Class*]

This class represents information about global functions. An instance of this class is returned by a call to `function-info` when it turns out that the function name passed as an argument refers to a global function.

⇒ `:name` [*Initarg*]

This initarg supplies the name of the global function. This initarg must be supplied.

⇒ `:type` [*Initarg*]

This initarg is used to supply the *type* of the global function. The type can be any function type specifier and it may contain user-defined types. If this initarg is omitted, it defaults to `t`.

⇒ `:inline` [*Initarg*]

This initarg is used to supply *inline* information about the global function. The value of this initarg can be either `inline` (i.e., the symbol with that name in the `common-lisp` package) meaning that the function is declared `inline`, `notinline` (i.e., the symbol with that name in the `common-lisp` package) meaning that the function is declared `notinline`, or `nil` meaning that no inline declaration for this function is in scope. If this initarg is not supplied, it defaults to `nil`.

⇒ `:ignore` [*Initarg*]

This initarg is used to supply *ignore* information about the global function. The value of this initarg can be either `ignore` (i.e., the symbol with that name in the `common-lisp` package) meaning that the function is declared `ignore`, `ignorable` (i.e., the symbol with that name in the `common-lisp` package) mean-

ing that the function is declared `ignorable`, or `nil` meaning that no ignore or ignorable declaration for this function is in scope. If this initarg is not supplied, it defaults to `nil`.

⇒ `:compiler-macro` [*Initarg*]

This initarg is used to supply a *compiler macro function* when a compiler macro is associated with the global function. If this initarg is not given, it defaults to `nil`, meaning that no compiler macro is associated with this function.

⇒ `name` *(info* `global-function-info`*)* [*Method*]

Given an instance of the class `global-function-info`, this method returns the name of the global function as supplied by the initarg `:name`.

⇒ `type` *(info* `global-function-info`*)* [*Method*]

Given an instance of the class `global-function-info`, this method returns the *type* of the global function as supplied by the initarg `:type`. If that initarg was not supplied, this method returns `t`.

⇒ `inline` *(info* `global-function-info`*)* [*Method*]

Given an instance of the class `global-function-info`, this method returns the *inline* information of the global function as supplied by the initarg `:inline`. If that initarg was not supplied, this method returns `nil`.

⇒ `ignore` *(info* `global-function-info`*)* [*Method*]

Given an instance of the class `global-function-info`, this method returns the *ignore* information of the global function as supplied by the initarg `:ignore`. If that initarg was not supplied, this method returns `nil`.

⇒ `compiler-macro` *(info* `global-function-info`*)* [*Method*]

Given an instance of the class `global-function-info`, this method returns the *compiler macro function* information associated with the global function, as supplied by the initarg `:compiler-macro`. If that initarg was not supplied, this method returns `nil`.

Local macro information

⇒ `local-macro-info` [*Class*]

This class represents information about local macros introduced by `macrolet`. An instance of this class is returned by a call to `function-info` when it turns out that the function name passed as an argument refers to a local macro.

⇒ `:name` [*Initarg*]

This initarg supplies the name of the local macro. This initarg must be supplied.

⇒ `:expander` [*Initarg*]

This initarg is used to supply the macro function used to expand macro forms that use this macro. This initarg must be supplied.

⇒ `name` *(info* `local-macro-info`*)* [*Method*]

Given an instance of the class `local-macro-info`, this method returns the name of the local macro as supplied by the initarg `:name`.

⇒ `expander` *(info* `local-macro-info`*)* [*Method*]

Given an instance of the class `local-macro-info`, this method returns the expander of the local macro as supplied by the initarg `:expander`.

Global macro information

⇒ `global-macro-info` [*Class*]

This class represents information about global macros introduced by `macrolet`. An instance of this class is returned by a call to `function-info` when it turns out that the function name passed as an argument refers to a global macro.

⇒ `:name` [*Initarg*]

This initarg supplies the name of the global macro. This initarg must be supplied.

⇒ `:expander` [*Initarg*]

This initarg is used to supply the macro function used to expand macro forms that use this macro. This initarg must be supplied.

⇒ `:compiler-macro` [*Initarg*]

This initarg is used to supply a *compiler macro function* when a compiler macro is associated with the global macro. If this initarg is not given, it defaults to `nil`, meaning that no compiler macro is associated with this macro.

⇒ `name` *(info* `global-macro-info`*)* [*Method*]

Given an instance of the class `global-macro-info`, this method returns the name of the global macro as supplied by the initarg `:name`.

⇒ `expander` *(info* `global-macro-info`*)* [*Method*]

Given an instance of the class `global-macro-info`, this method returns the expander of the global macro as supplied by the initarg `:expander`.

⇒ `compiler-macro` *(info* `global-macro-info`*)* [*Method*]

Given an instance of the class `global-macro-info`, this method returns the *compiler macro function* information associated with the global macro, as supplied by the initarg `:compiler-macro`. If that initarg was not supplied, this method returns `nil`.

Special operator information

⇒ `special-operator-info` [*Class*]

This class represents information about special operators. An instance of this class is returned by a call to `function-info` when it turns out that the function name passed as an argument refers to a specialoperator.

⇒ `:name` [*Initarg*]

This initarg supplies the name of the special operator. This initarg must be supplied.

⇒ `name` *(info* `special-operator-info`*)* [*Method*]

Given an instance of the class `special-operator-info`, this method returns the name of the special operator as supplied by the initarg `:name`.

2.3.3 Block information

⇒ `block-info` *environment symbol* [*Generic Function*]

⇒ `no-block-info` [*Condition*]

This condition is signaled by Cleavir when a client-supplied method on the generic function `block-info` returns `nil`.

⇒ `name` *(condition* `no-block-info`*)* [*Method*]

This method returns the name of the block for which no info was available.

⇒ `block-info` [*Class*]

This class represents information about blocks introduced by `block`. An instance of this class is returned by a call to `block-info` when the symbol passed as an argument refers to a block.

⇒ `:name` [*Initarg*]

This initarg supplies the name of the block. This initarg must be supplied.

⇒ `:identity` [*Initarg*]

This initarg is used to supply some kind of implementation-defined *identity*. The implementation can supply any object as the identity, because it is not interpreted by the compiler. However, the *same* identity must be supplied each time for a particular block. This initarg must be supplied.

⇒ `name` *(info* `block-info`*)* [*Method*]

Given an instance of the class `block-info`, this method returns the name of the block as supplied by the initarg `:name`.

⇒ `identity` *(info* `block-info`*)* [*Method*]

Given an instance of the class `block-info`, this method returns the identity of the block as supplied by the initarg `:identity`.

2.3.4 Tag information

⇒ `tag-info` *environment tag* [*Generic Function*]

⇒ `no-tag-info` [*Condition*]

This condition is signaled by Cleavir when a client-supplied method on the generic function `tag-info` returns `nil`.

⇒ `name` *(condition* `no-tag-info`*)* [*Method*]

This method returns the name of the tag for which no info was available.

⇒ `tag-info` [*Class*]

⇒ `:name` [*Initarg*]

This initarg supplies the name of the tag. This initarg must be supplied.

⇒ `:identity` [*Initarg*]

This initarg is used to supply some kind of implementation-defined *identity*. The implementation can supply any object as the identity, because it is not interpreted by the compiler. However, the *same* identity must be supplied each time for a particular tag. This initarg must be supplied.

⇒ `name` *(info* `tag-info`*)* [*Method*]

Given an instance of the class `tag-info`, this method returns the name of the block as supplied by the initarg `:name`.

⇒ `identity` *(info* `tag-info`*)* [*Method*]

Given an instance of the class `tag-info`, this method returns the identity of the tag as supplied by the initarg `:identity`.

2.3.5 Optimization information

⇒ `optimize-info` *environment* [*Generic Function*]

Client-supplied methods on this function must always return a valid instance of the class `optimize-info`. For qualities that have no explicit entries in the environment, the default values should be supplied.

⇒ `optimize-info` [*Class*]

⇒ `:speed` [*Initarg*]

This initarg supplies a value for the *speed* quality. The argument must be an integer between 0 and 3 inclusive. If this initarg is omitted, the value defaults to 3.

⇒ `:compilation-speed` [*Initarg*]

This initarg supplies a value for the *compilation-speed* quality. The argument must be an integer between 0 and 3 inclusive. If this initarg is omitted, the value defaults to 3.

⇒ `:safety` [*Initarg*]

This initarg supplies a value for the *safety* quality. The argument must be an integer between 0 and 3 inclusive. If this initarg is omitted, the value defaults to 3.

⇒ `:debug` [*Initarg*]

This initarg supplies a value for the *debug* quality. The argument must be an integer between 0 and 3 inclusive. If this initarg is omitted, the value defaults to 3.

⇒ `:space` [*Initarg*]

This initarg supplies a value for the *space* quality. The argument must be an integer between 0 and 3 inclusive. If this initarg is omitted, the value defaults to 3.

⇒ `speed` *(info* `optimize-info`*)* [*Method*]

This method returns the value of the *speed* quality as supplied by the initarg `:speed`. If that initarg was not supplied the value 3 is returned.

⇒ `compilation-speed` *(info* `optimize-info`*)* [*Method*]

This method returns the value of the *compilation-speed* quality as supplied by the initarg `:compilation-speed`. If that initarg was not supplied the value 3 is returned.

⇒ `safety` *(info* `optimize-info`*)* [*Method*]

This method returns the value of the *safety* quality as supplied by the initarg `:safety`. If that initarg was not supplied the value 3 is returned.

⇒ `debug` *(info* `optimize-info`*)* [*Method*]

This method returns the value of the *debug* quality as supplied by the initarg `:debug`. If that initarg was not supplied the value 3 is returned.

⇒ `space` *(info* `optimize-info`*)* [*Method*]

This method returns the value of the *space* quality as supplied by the initarg `:space`. If that initarg was not supplied the value 3 is returned.

2.3.6 Type information

⇒ `type-expand` *environment type* [*Generic Function*]

This generic function is called with some arbitrary type specifier, and the return value is an equivalent type specifier that does not contain any user-defined types introduced by `deftype`.

2.4 Augmenting the environment

2.4.1 Adding lexical variables

⇒ `add-lexical-variable` *environment symbol* [*Generic Function*]

This generic function is called by the compiler in order to add *symbol* as the name of a lexical variable to *environment*. The function returns a new environment augmented with the new information. The original environment is not affected.

2.4.2 Adding special variables

⇒ `add-special-variable` *environment symbol* [*Generic Function*]

This generic function is called by the compiler in order to add *symbol* as the name of a special variable to *environment*. The function returns a new environment augmented with the new information. The original environment is not affected.

2.4.3 Adding local symbol macros

⇒ `add-local-symbol-macro` *environment symbol expansion* [*Generic Function*]

This generic function is called by the compiler in order to add a local symbol macro to *environment*. The parameter *symbol* is the name of the macro and the parameter *expansion* is the expansion to be associated with the name. The function returns a new environment augmented with the new information. The original environment is not affected.

2.4.4 Adding local functions

⇒ `add-local-function` *environment function-name* [*Generic Function*]

This generic function is called by the compiler in order to add *function-name* as the name of a local function to *environment*. The function returns a new environment augmented with the new information. The original environment is not affected.

2.4.5 Adding local macros

⇒ `add-local-macro` *environment symbol expander* [*Generic Function*]

This generic function is called by the compiler in order to add a local macro to *environment*. The parameter *symbol* is the name of the macro and the parameter *expander* is a *macro function* to be associated with the name. The function returns a new environment augmented with the new information. The original environment is not affected.

2.4.6 Adding blocks

⇒ `add-block` *environment symbol* [*Generic Function*]

This generic function is called by the compiler in order to add *symbol* as the name of a block to *environment*. The function returns a new environment augmented with the new information. The original environment is not affected.

2.4.7 Adding tags

⇒ `add-tag` *environment tag* [*Generic Function*]

This generic function is called by the compiler in order to add *tag* as the name of a tag to *environment*. The tag is either a symbol or an integer. The function returns a new environment augmented with the new information. The original environment is not affected.

2.4.8 Adding type information

⇒ `add-variable-type` *environment symbol type* [*Generic Function*]

This generic function is called by the compiler in order to add type information about a variable. The compiler has already verified that there is a lexical variable, a special variable, or a symbol macro named *symbol* in scope in *environment*. The function returns a new environment augmented with the new information. The original environment is not affected.

⇒ `add-function-type` *environment function-name type* [*Generic Function*]

This generic function is called by the compiler in order to add type information about a function. The compiler has already verified that there is a function named *function-name* in scope in *environment*. The function returns a new environment augmented with the new information. The original environment is not affected.

2.4.9 Adding ignore information

⇒ `add-variable-ignore` *environment symbol ignore* [*Generic Function*]

This generic function is called by the compiler in order to add ignore information about a variable. The compiler has already verified that there is a lexical variable or a special variable named *symbol* in scope in *environment*. The value of the *ignore* parameter is either the symbol `ignore` or the symbol `ignorable`. The function returns a new environment augmented with the new information. The original environment is not affected.

⇒ `add-function-ignore` *environment function-name ignore* [*Generic Function*]

This generic function is called by the compiler in order to add ignore information about a function. The compiler has already verified that there is a local function or a global function named *function-name* in scope in *environment*. The value of the *ignore* parameter is either the symbol `ignore` or the symbol `ignorable`. The function returns a new environment augmented with the new information. The original environment is not affected.

2.4.10 Adding dynamic-extent information

⇒ `add-variable-dynamic-extent` *environment symbol* [*Generic Function*]

This generic function is called by the compiler in order to add dynamic-extent information about a variable. The compiler has already verified that there is a lexical variable named *symbol* in scope in *environment*. The function returns a new environment augmented with the new information. The original environment is not affected.

⇒ `add-function-dynamic-extent` *environment function-name* [*Generic Function*]

This generic function is called by the compiler in order to add dynamic-extent information about a function. The compiler has already verified that there is a local function named *function-name* in scope in *environment*. The function returns a new environment augmented with the new information. The original environment is not affected.

2.4.11 Adding optimize information

⇒ `add-optimize` *environment quality value* [*Generic Function*]

This generic function is called by the compiler in order to add optimize information to the environment. The value of the *quality* parameter is one of the symbols `speed`, `compilation-speed`, `debug`, `safety`, and `space`. The value of the *value* parameter is an integer between 0 and 3 inclusive. The function returns a new environment augmented with the new information. The original environment is not affected.

2.4.12 Adding inline information

⇒ `add-inline` *environment function-name inline* [*Generic Function*]

This generic function is called by the compiler in order to add inline information about a function. The compiler has already verified that there is a local function or a global function named *function-name* in scope in *environment*. The value of the *inline* parameter is either the symbol `inline` or the symbol `notinline`. The function returns a new environment augmented with the new information. The original environment is not affected.

2.5 Compile-time evaluation

⇒ `eval` *environment form* [*Generic Function*]

This function is called by the compiler in order to evaluate forms in *environment*. It is called as a result of top-level forms being wrapped in `eval-when :compile-toplevel`.

2.6 Default augmentation classes

⇒ `entry` [*Class*]

This class is the base class of all entries in this section.

2.6.1 Lexical variable

⇒ `lexical-variable` [*Class*]

This class represents a lexical variable. It is a subclass of the class `entry`

⇒ `:name` [*Initarg*]

This initarg is used to specify the name of the lexical variable. The value must be a symbol. This initarg must be supplied.

⇒ `name` *(entry* `lexical-variable`*)* [*Method*]

Given an instance of the class `lexical-variable`, this method returns the name of the lexical variable as supplied by the initarg `:name`.

2.6.2 Special variable

⇒ `special-variable` [*Class*]

⇒ `:name` [*Initarg*]

This initarg is used to specify the name of the special variable. The value must be a symbol. This initarg must be supplied.

⇒ `:global-p` [*Initarg*]

This initarg indicates whether the special variable is globally proclaimed as special. If omitted, the default value is `nil`.

⇒ `name` *(entry* `special-variable`*)* [*Method*]

Given an instance of the class `special-variable`, this method returns the name of the special variable as supplied by the initarg `:name`.

⇒ `global-p` *(entry* `special-variable`*)* [*Method*]

2.6.3 Symbol macro

⇒ `symbol-macro` [*Class*]

⇒ `:name` [*Initarg*]

This initarg is used to specify the name of the symbol macro. The value must be a symbol. This initarg must be supplied.

⇒ `:expansion` [*Initarg*]

⇒ `name` *(entry* `symbol-macro`*)* [*Method*]

Given an instance of the class `symbol-macro`, this method returns the name of the symbol macro as supplied by the initarg `:name`.

⇒ `expansion` *(entry* `symbol-macro`*)* [*Method*]

2.6.4 Function

⇒ `function` [*Class*]

⇒ `:name` [*Initarg*]

This initarg is used to specify the name of the function. The value must be a function name. This initarg must be supplied.

⇒ `:identity` [*Initarg*]

⇒ `name` *(entry* `function`*)* [*Method*]

Given an instance of the class `function`, this method returns the name of the function as supplied by the initarg `:name`.

⇒ `identity` *(entry* `function`*)* [*Method*]

2.6.5 Macro

⇒ `macro` [*Class*]

⇒ `:name` [*Initarg*]

This initarg is used to specify the name of the macro. The value must be a symbol. This initarg must be supplied.

⇒ `:expander` [*Initarg*]

⇒ `name` *(entry* `macro`*)* [*Method*]

Given an instance of the class `macro`, this method returns the name of the macro as supplied by the initarg `:name`.

⇒ `expander` *(entry* `macro`*)* [*Method*]

2.6.6 Block

⇒ `block` [*Class*]

⇒ `:name` [*Initarg*]

This initarg is used to specify the name of the block. The value must be a

symbol. This initarg must be supplied.

⇒ `:identity` [*Initarg*]

⇒ `name` *(entry* `block`*)* [*Method*]

⇒ `identity` *(entry* `block`*)* [*Method*]

2.6.7 Tag

⇒ `tag` [*Class*]

⇒ `:name` [*Initarg*]

This initarg is used to specify the name of the tag. The value must be a go tag, i.e. a symbol or an integer. This initarg must be supplied.

⇒ `:identity` [*Initarg*]

⇒ `name` *(entry* `tag`*)* [*Method*]

⇒ `identity` *(entry* `tag`*)* [*Method*]

2.6.8 Variable type

⇒ `variable-type` [*Class*]

⇒ `:name` [*Initarg*]

This initarg is used to specify the name of the variable. The value must be a symbol. This initarg must be supplied.

⇒ `:type` [*Initarg*]

⇒ `name` *(entry* `variable-type`*)* [*Method*]

⇒ `type` *(entry* `variable-type`*)* [*Method*]

2.6.9 Function type

⇒ `function-type` [*Class*]

⇒ `:name` [*Initarg*]

This initarg is used to specify the name of the function. The value must be a function name. This initarg must be supplied.

⇒ `:type` [*Initarg*]

⇒ `name` *(entry* `function-type`*)* [*Method*]

⇒ `type` *(entry* `function-type`*)* [*Method*]

2.6.10 Variable ignore

⇒ `variable-ignore` [*Class*]

⇒ `:name` [*Initarg*]

This initarg is used to specify the name of the variable. The value must be a symbol. This initarg must be supplied.

⇒ `:ignore` [*Initarg*]

⇒ `name` *(entry* `variable-ignore`*)* [*Method*]

⇒ `ignore` *(entry* `variable-ignore`*)* [*Method*]

2.6.11 Function ignore

⇒ `function-ignore` [*Class*]

⇒ `:name` [*Initarg*]

This initarg is used to specify the name of the function. The value must be a function name. This initarg must be supplied.

⇒ `:ignore` [*Initarg*]

⇒ `name` *(entry* `function-ignore`*)* [*Method*]

⇒ `ignore` *(entry* `function-ignore`*)* [*Method*]

2.6.12 Variable dynamic extent

⇒ `variable-dynamic-extent` [*Class*]

⇒ `:name` [*Initarg*]

This initarg is used to specify the name of the lexical variable. The value must be a symbol. This initarg must be supplied.

⇒ `name` *(entry* `variable-dynamic-extent`*)* [*Method*]

2.6.13 Function dynamic-extent

⇒ `function-dynamic-extent` [*Class*]

⇒ `:name` [*Initarg*]

This initarg is used to specify the name of the function. The value must be a function name. This initarg must be supplied.

⇒ `name` *(entry* `function-dynamic-extent`*)* [*Method*]

2.6.14 Inline

⇒ `inline` [*Class*]

⇒ `:name` [*Initarg*]

This initarg is used to specify the name of the function. The value must be a function name. This initarg must be supplied.

⇒ `:inline` [*Initarg*]

⇒ `name` *(entry* `inline`*)* [*Method*]

⇒ `inline` *(entry* `inline`*)* [*Method*]

2.6.15 Optimize

⇒ `optimize` [*Class*]

⇒ `:quality` [*Initarg*]

⇒ `:value` [*Initarg*]

⇒ `quality` *(entry* `optimize`*)* [*Method*]

⇒ `value` *(entry* `optimize`*)* [*Method*]

Chapter 3

Primitive operations

3.1 Purpose

Primitive operations (or primops for short) are similar to Common Lisp special operators in that the compiler handles them specially. They are different from Common Lisp special operators in that a primop does not necessarily have an evaluation rule that is different from that of a function call.

Generally speaking, primops should not be used directly in application code. Instead, they are used in system code for implementing certain basic Common Lisp function. So, for example, the `car` primop would typically be used only in the code for the Common Lisp function `car`. That function would then be inlined by the compiler, so that the resulting AST and ultimately the resulting HIR instruction that the `car` primop translates to will be present also in application code.

Frequently, the need for a primop comes from arises because some HIR instruction is needed. Take, for example, the HIR instruction named `eq-instruction` for comparing two pointer values for equality. The existence of that instruction requires an AST doing the same thing, and it is called `eq-ast`. When compiled to HIR, the `eq-ast` generates the an `eq-instruction`. Finally, in order to produce the source for the Common Lisp function `eq`, it must be possible to produce the `eq-ast` from some Common Lisp code, which is why the `eq`

primop is needed.

3.2 Package

All primitive operations have names that are symbols in the package named `cleavir-primop`. This package contains only those symbols that name the primitive operations. There is no code directly associated with the primitive operations. Instead, the code for translating primitive operations to abstract syntax trees (See Chapter 4.) is in the form of methods on the function `cleavir-generate-ast:convert-special`.

3.3 Existing primitive operations

⇒ `eq` [*Primitive operation*]

This primitive operation has the same semantics as the Common Lisp function `eq`, except that it can only appear as the *test-form* in the special form `if`. Its main purpose is for defining the code for that Common Lisp function. Typically the Common Lisp `eq` function will be declared `inline` so that the abstract syntax tree and HIR instruction resulting from this primop will eventually end up in the compiled code of many applications.

⇒ `typeq` [*Primitive operation*]

This primitive operation is similar to the Common Lisp function `typep`. It differs from the Common Lisp function in that it does not evaluate its second argument, i.e., the type specifier. Also, it does not take an optional environment argument[1] like `typep` does.

The implementation may have a compiler macro on the function `typep` so that it turns into `typeq` when the second argument is a constant. The `typeq`

[1] We may have to add an environment argument to `typeq`. Alternatively we may have to specify that it only takes certain types that do not depend on the environment.

primitive operation can also be used in the implementation of certain system functions that need to check for constant types.

The `typeq` generates a `typeq` AST and ultimately a `typeq` HIR instruction. This instruction is used by the type inferencer (See Section 8.4.) to infer different types in different successor branches.

⇒ `car` [*Primitive operation*]

This primitive operation is typically used in the implementation of the Common Lisp function `car`. The main difference between this primitive operation and the Common Lisp function is that the primitive operation requires its argument to evaluate to a `cons`. For that reason, if used to implement the Common Lisp function `car`, this primitive operation should be preceded by a test (using the primitive operation `typeq`) to verify that the argument is a `cons`. A typical implementation of the `car` function might look like this:

```
(defun car (object)
  (if (cleavir-primop:typeq object cons)
      (cleavir-primop:car object)
      (if (cleavir-primop:typeq object null)
          nil
          (error 'type-error
                 :datum object
                 :expected-type '(or cons null)))))
```

The Common Lisp `car` function will typically be declared `inline`, allowing the type inferencer (See Section 8.4.) to use the `typeq` primitive operation to remove redundant type checks.

⇒ `cdr` [*Primitive operation*]

This primitive operation is typically used in the implementation of the Common Lisp function `cdr`. The main difference between this primitive operation and the Common Lisp function is that the primitive operation requires its argument to evaluate to a `cons`. For that reason, if used to implement the Common Lisp function `cdr`, this primitive operation should be preceded by a test (using the

primitive operation `typeq`) to verify that the argument is a `cons`. A typical implementation of the `cdr` function might look like this:

```
(defun cdr (object)
  (if (cleavir-primop:typeq object cons)
      (cleavir-primop:cdr object)
      (if (cleavir-primop:typeq object null)
          nil
          (error 'type-error
                 :datum object
                 :expected-type '(or cons null)))))
```

The Common Lisp `cdr` function will typically be declared `inline`, allowing the type inferencer (See Section 8.4.) to use the `typeq` primitive operation to remove redundant type checks.

⇒ `rplaca` [*Primitive operation*]

This primitive operation is typically used in the implementation of the Common Lisp function `rplaca`. It requires its first argument to be a `cons`. For that reason, if used to implement the Common Lisp function `rplaca`, this primitive operation should be preceded by a test (using the primitive operation `typeq`) to verify that the argument is a `cons`. A typical implementation of the `rplaca` function might look like this:

```
(defun rplaca (cons object)
  (if (cleavir-primop:typeq object cons)
      (cleavir-primop:rplaca cons object)
      (error 'type-error
             :datum object
             :expected-type '(or cons null))))
```

⇒ `rplacd` [*Primitive operation*]

This primitive operation is typically used in the implementation of the Common Lisp function `rplacd`. It requires its first argument to be a `cons`. For that

reason, if used to implement the Common Lisp function `rplacd`, this primitive operation should be preceded by a test (using the primitive operation `typeq`) to verify that the argument is a `cons`. A typical implementation of the `rplacd` function might look like this:

```
(defun rplacd (cons object)
  (if (cleavir-primop:typeq object cons)
      (cleavir-primop:rplacd cons object)
      (error 'type-error
             :datum object
             :expected-type '(or cons null))))
```

Chapter 4

Abstract syntax tree

4.1 Purpose

The abstract syntax tree[1] (AST for short) is a structured version of the source code in which the use for an environment has been eliminated.[2] An abstract syntax tree consists of a graph of class instances. Each class corresponds to some intrinsic category of source code elements.

Every lexical variable has been replaced by a class instance that contains its name, but different variables with the same name in the source code are represented by different instances. Global functions and special variables are similarly represented by class instances that contain the name of the function or variable. Since different lexical variables with different names are represented by different instances, there is not need for scoping operators such as `let` or `let*`, which is why there are no classes that correspond to these source constructs.

Macros and symbol macros have been expanded so that the only operators that remain are special operators and global functions.

[1] An abstract syntax tree is not a tree, and not even an acyclic graph, but it is traditionally called an abstract syntax tree anyway, so we keep this terminology.

[2] More traditional languages use the term *abstract syntax tree* for a notation that is much closer to the source code than the notation we use here.

There are no classes that correspond to declarations. Implementations should replace type declarations by the use of instances of the `the-ast` class or the `typeq-ast` class as indicated by the HyperSpec

Functions that can not be expressed as simpler functions, and that are performance critical, have been given their own abstract syntax trees. For instance the function that given a `cons` cell returns the `car` of that `cons` cell (this is only part of what the Common Lisp function `car` does) has been given its own abstract syntax tree class. Implementations are under no obligation to use these classes. It is perfectly possible for an implementation to represent every function call to a standard Common Lisp function by a `call-ast`. However, Cleavir contains features that make it easier to optimize the resulting code if these classes are used.

Implementations are free to create subclasses of the AST classes listed here, for instance in order to add additional information to AST nodes. In that case, it might be necessary to provide overriding or extending methods on the generic functions that handle the abstract syntax tree, in particular when it is translated into MIR.

Implementations are also free to create entirely new AST classes. Then it might be necessary to provide primary methods on some generic functions that manipulate the abstract syntax tree.[3]

4.2 General types of abstract syntax trees

There is no abstract syntax tree class corresponding to the special operator `eval-when`. The compiler takes action based on the context of the compilation and either evaluates or generates an abstract syntax tree.

4.2.1 `immediate-ast`

Initarg	Reader	Description
`:value`	`value`	The immediate value.

[3] FIXME: At some point, provide a list of generic functions that would require overriding, extending, or primary methods.

A `immediate-ast` is used by an implementation to represent constants that can be represented as immediates in machine code on the particular platform used.

4.2.2 `constant-ast`

Initarg	Reader	Description
`:value`	`value`	The corresponding constant.

A `constant-ast` is typically used by an implementation to represent constants in the source code.

This constant can either be in the form of a self-evaluating object, the value of a constant variable, or a quoted object.

At this point, there is no discrimination between the different types of constant, nor between integers of different magnitude. Later on in the compilation process, some backend may replace small integer constants by immediate values.

4.2.3 `fdefinition-ast`

Initarg	Reader	Description
`:name-ast`	`name-ast`	The name of the function.

The `fdefinition-ast` is produced whenever the compiler sees a reference to a *global function*, i.e., a name in a functional position that is not defined as a local function, as a special operator, nor as a macro. It can also be produced as an inline version of the function `fdefinition`.

The `name-ast` is an AST that produces the name of function. It can be either a `constant-ast`, in which case the constant is a function name, or it can be some other AST that evaluates to the name of a function. When the `fdefinition-ast` is produced as a result of a name in the functional position of a form, then the `name-ast` is always a `constant-ast`.

4.2.4 `symbol-value-ast`

Initarg	Reader	Description
`:symbol-ast`	`symbol-ast`	The name of the variable.

A `symbol-value-ast` is produced whenever the compiler sees a reference to a *special variable*, i.e. a name in a variable position that is not defined as a lexical variable, a constant variable, nor as a symbol macro. It can also be produced as an inline version of the function `symbol-value`.

The `symbol-ast` is an AST that produces the name of the variable. It can be either a `constant-ast`, in which case the constant is a symbol, or it can be some other AST that evaluates to a symbol. When the `symbol-value-ast` is produced as a result of a reference to a special variable, then the `symbol-ast` is always a `constant-ast`.

4.2.5 `set-symbol-value-ast`

Initarg	Reader	Description
`:symbol-ast`	`symbol-ast`	The name of the variable.
`:value-ast`	`value-ast`	The value to be assigned.

A `set-symbol-value-ast` is produced whenever the compiler sees an assignment to a *special variable*. It can also be produced as an inline version of the function `(setf symbol-value)`.

The `symbol-ast` is an AST that produces the name of the variable. It can be either a `constant-ast`, in which case the constant is a symbol, or it can be some other AST that evaluates to a symbol. When the `symbol-value-ast` is produced as a result of an assignment to a special variable, then the `symbol-ast` is always a `constant-ast`.

The `value-ast` is an AST that evaluates to the value to be assigned to the variable.

This AST does not produce any value. It must appear in a context where no value is required, such as one of the forms preceding the last form of a `progn-ast` (See Section 4.2.13.).

4.2.6 `lexical-ast`

Initarg	Reader	Description
`:name`	`name`	The name of the variable.

A `lexical-ast` is produced whenever the compiler sees a reference to a *lexical variable*.

4.2.7 `call-ast`

Initarg	Reader	Description
`:callee-ast`	`callee-ast`	The AST of the function to be called
`:argument-asts`	`argument-asts`	A list of ASTs, one for each argument to the function.

A `call-ast` is produced whenever the compiler sees a function call. It contains ASTs for the function to be called and for the arguments to that function. The AST representing the function can be any AST that produces a function as a value.

4.2.8 `block-ast`

Initarg	Reader	Description
`:body-ast`	`body-ast`	The AST of the body.

The `block-ast` is used to represent the Common Lisp special operator `block`. The `block-ast` does not contain the name of the block because the `return-from-ast` contains a reference to the corresponding `block-ast`.

4.2.9 `function-ast`

Initarg	Reader	Description
`:body-ast`	`body-ast`	The AST for the body of the function.
`:lambda-list`	`lambda list`	The lambda list of the function.

The `function-ast` is one of the more complicated ones, because it hides all

the implementation details of how arguments are parsed.

4.2.10 `go-ast`

Initarg	Reader	Description
`:tag-ast`	`body-ast`	The AST for the tag.

4.2.11 `if-ast`

Initarg	Reader	Description
`:test-ast`	`body-ast`	The AST for the test.
`:then-ast`	`then-ast`	The AST for the *then* branch.
`:else-ast`	`else-ast`	The AST for the *else* branch.

The `if-ast` corresponds directly to the Common Lisp special operator `if`.

4.2.12 `load-time-value-ast`

Initarg	Reader	Description
`:form-ast`	`form-ast`	AST for the form.

A `load-time-value-ast` is produced whenever the compiler sees a `load-time-value` special form, but also when it sees a *constant* and processing is done by `compile-file`, rather than by `compile` or `eval`.

This AST always contains a *form*, so that if it is the result of a *constant*, then it is wrapped in a `quote` special form.

4.2.13 `progn-ast`

Initarg	Reader	Description
`:form-asts`	`form-asts`	A list of ASTs of the body.

The `progn-ast` corresponds directly to the Common Lisp special operator `progn`. It returns the values that are return by the last AST in the child

forms.

4.2.14 `return-from-ast`

Initarg	Reader	Description
`:block-ast`	`block-ast`	The AST of the corresponding block.
`:form-ast`	`form-ast`	The AST of the value form.

4.2.15 `setq-ast`

Initarg	Reader	Description
`:lhs-ast`	`lhs-ast`	The AST of the left-hand side.
`:value-ast`	`value-ast`	The AST of the value form.

This AST does not produce any value. It must appear in a context where no value is required, such as one of the forms preceding the last form of a `progn-ast` (See Section 4.2.13.).

4.2.16 `tagbody-ast`

Initarg	Reader	Description
`:items`	`items`	A list of ASTs corresponding to the items of the `tagbody` form.

An item can either be a `tag-ast` or an AST corresponding to some `statement`.

4.2.17 `tag-ast`

Initarg	Reader	Description
`:name`	`name`	The name of the tag.

4.2.18 `the-ast`

Initarg	Reader	Description
`:type-specifiers`	`type-specifiers`	The second argument.
`:form-ast`	`form-ast`	The first argument.

This AST should only be used in unsafe code, or in situations where the compiler needs to be informed that a variable has a particular type. In safe code, the `typeq-ast` (See Section 4.2.19.) should be used instead. The effect of this AST is to promise that the value of its child has a particular type, without checking this fact.

4.2.19 `typeq-ast`

Initarg	Reader	Description
`:type-ast`	`type-ast`	The second argument.
`:variable-ast`	`variable-ast`	The first argument.

4.3 Abstract syntax trees for fixnum arithmetic

4.3.1 `fixnum-add-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.
`:result`	`result`	The result of the operation.

This AST can be used by implementations that represent fixnums in a way that allows them to be added directly without first unboxing them.

This AST can only occur as the first child of an `if-ast` (See Section 4.2.11.). It has three children.

The first and the second child are ASTs that represent the arguments to the fixnum operation. The third child is a `lexical-ast` that will hold the result of the operation.

The semantics of this AST are that the operand are added and if the outcome is *normal*, i.e., there is no overflow, then the lexical variable is assigned the result of the operation, i.e., the sum of the two operands. If the operation results in an *overflow*, then the lexical variable is still a fixnum representing the result of the operation as follows: If the result is *negative*, then the sum of the two operands is $2^n + r$ where n is the number of bits used to represent a fixnum, and r is the value of the lexical variable. If the result is *positive*, then the sum of the two operands is $r - 2^n$.

4.3.2 `fixnum-sub-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.
`:result`	`result`	The result of the operation.

This AST can be used by implementations that represent fixnums in a way that allows them to be subtracted directly without first unboxing them.

This AST can only occur as the first child of an `if-ast` (See Section 4.2.11.). It has three children.

The first and the second child are ASTs that represent the arguments to the fixnum operation. The third child is a `lexical-ast` that will hold the result of the operation.

The semantics of this AST are that the operand are subtracted and if the outcome is *normal*, i.e., there is no overflow, then the lexical variable is assigned the result of the operation, i.e., the difference between the two operands. If the operation results in an *overflow*, then the lexical variable is still a fixnum representing the result of the operation as follows: If the result is *negative*, then the difference between the two operands is $2^n + r$ where n is the number of bits used to represent a fixnum, and r is the value of the lexical variable. If the result is *positive*, then the difference between the two operands is $r - 2^n$.

4.3.3 `fixnum-less-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementations that represent fixnums in a way that allows them to be compared directly without first unboxing them.

This AST can only occur as the first child (i.e., the *test*) of an `if-ast` (See Section 4.2.11.). It has two children that represent the arguments to the comparison.

Semantically, this AST returns *true* if an only if the first argument is strictly less than the second (and *false* otherwise), except that since this AST can only occur in the test position of an `if-ast`, no value will ever be returned. Instead this AST translates to a test and a branch.

4.3.4 `fixnum-not-less-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementations that represent fixnums in a way that allows them to be compared directly without first unboxing them.

This AST can only occur as the first child (i.e., the *test*) of an `if-ast` (See Section 4.2.11.). It has two children that represent the arguments to the comparison.

Semantically, this AST returns *true* if an only if the first argument is not strictly less than the second (and *false* otherwise), except that since this AST can only occur in the test position of an `if-ast`, no value will ever be returned. Instead this AST translates to a test and a branch.

4.3.5 `fixnum-greater-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementations that represent fixnums in a way that allows them to be compared directly without first unboxing them.

This AST can only occur as the first child (i.e., the *test*) of an `if-ast` (See Section 4.2.11.). It has two children that represent the arguments to the comparison.

Semantically, this AST returns *true* if an only if the first argument is strictly greater than the second (and *false* otherwise), except that since this AST can only occur in the test position of an `if-ast`, no value will ever be returned. Instead this AST translates to a test and a branch.

4.3.6 `fixnum-not-greater-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementations that represent fixnums in a way that allows them to be compared directly without first unboxing them.

This AST can only occur as the first child (i.e., the *test*) of an `if-ast` (See Section 4.2.11.). It has two children that represent the arguments to the comparison.

Semantically, this AST returns *true* if an only if the first argument is not strictly greater than the second (and *false* otherwise), except that since this AST can only occur in the test position of an `if-ast`, no value will ever be returned. Instead this AST translates to a test and a branch.

4.3.7 `fixnum-equal-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementations that represent fixnums in a way that allows them to be compared directly without first unboxing them.

This AST can only occur as the first child (i.e., the *test*) of an `if-ast` (See Section 4.2.11.). It has two children that represent the arguments to the comparison.

Semantically, this AST returns *true* if an only if the first argument is equal to the second (and *false* otherwise), except that since this AST can only occur in the test position of an `if-ast`, no value will ever be returned. Instead this AST translates to a test and a branch.

4.4 Abstract syntax trees for floating-point arithmetic

The ASTs in this section can be used by implementation that want to inline floating-point arithmetic. There are four families of ASTs, one for each type of floating-point precision defined by the HyperSpec.

4.4.1 `short-float-add-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementation that support the short-float floating-point data type.

The arguments of this AST must evaluate to short-float values. The value of this AST is the short-float value representing the sum of the two arguments.

4.4.2 `short-float-sub-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementation that support the short-float floating-point data type.

The arguments of this AST must evaluate to short-float values. The value of this AST is the short-float value representing the difference of the two arguments.

4.4.3 `short-float-mul-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementation that support the short-float floating-point data type.

The arguments of this AST must evaluate to short-float values. The value of this AST is the short-float value representing the product of the two arguments.

4.4.4 `short-float-div-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementation that support the short-float floating-point data type.

The arguments of this AST must evaluate to short-float values. The value of this AST is the short-float value representing the quotient of the two arguments.

4.4.5 `short-float-less-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementation that support the short-float floating-point data type.

The arguments of this AST must evaluate to short-float values. The value is a Boolean indicating whether the first argument is strictly less than the second argument.

4.4.6 `short-float-not-greater-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementation that support the short-float floating-point data type.

The arguments of this AST must evaluate to short-float values. The value is a Boolean indicating whether the first argument is less than or equal to the second argument.

4.4.7 `short-float-greater-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementation that support the short-float floating-point data type.

The arguments of this AST must evaluate to short-float values. The value is a Boolean indicating whether the first argument is strictly greater than the

second argument.

4.4.8 `short-float-not-less-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementation that support the short-float floating-point data type.

The arguments of this AST must evaluate to short-float values. The value is a Boolean indicating whether the first argument is greater than or equal to the second argument.

4.4.9 `short-float-equal-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementation that support the short-float floating-point data type.

The arguments of this AST must evaluate to short-float values. The value is a Boolean indicating whether the first argument is equal to the second argument.

4.4.10 `single-float-add-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

The arguments of this AST must evaluate to single-float values. The value of this AST is the single-float value representing the sum of the two arguments.

4.4.11 `single-float-sub-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

The arguments of this AST must evaluate to single-float values. The value of this AST is the single-float value representing the difference of the two arguments.

4.4.12 `single-float-mul-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

The arguments of this AST must evaluate to single-float values. The value of this AST is the single-float value representing the product of the two arguments.

4.4.13 `single-float-div-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

The arguments of this AST must evaluate to single-float values. The value of this AST is the single-float value representing the quotient of the two arguments.

4.4.14 `single-float-less-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

The arguments of this AST must evaluate to single-float values. The value is a Boolean indicating whether the first argument is strictly less than the second argument.

4.4.15 `single-float-not-greater-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

The arguments of this AST must evaluate to single-float values. The value is a Boolean indicating whether the first argument is less than or equal to the second argument.

4.4.16 `single-float-greater-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

The arguments of this AST must evaluate to single-float values. The value is a Boolean indicating whether the first argument is strictly greater than the second argument.

4.4.17 `single-float-not-less-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

The arguments of this AST must evaluate to single-float values. The value is a Boolean indicating whether the first argument is greater than or equal to the second argument.

4.4.18 `single-float-equal-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

The arguments of this AST must evaluate to single-float values. The value is a Boolean indicating whether the first argument is equal to the second argument.

4.4.19 `double-float-add-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementation that support the double-float floating-point data type.

The arguments of this AST must evaluate to double-float values. The value of this AST is the double-float value representing the sum of the two arguments.

4.4.20 `double-float-sub-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementation that support the double-float floating-point data type.

The arguments of this AST must evaluate to double-float values. The value of this AST is the double-float value representing the difference of the two arguments.

4.4.21 `double-float-mul-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementation that support the double-float floating-point data type.

The arguments of this AST must evaluate to double-float values. The value of this AST is the double-float value representing the product of the two arguments.

4.4.22 `double-float-div-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementation that support the double-float floating-point data type.

The arguments of this AST must evaluate to double-float values. The value of this AST is the double-float value representing the quotient of the two arguments.

4.4.23 `double-float-less-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementation that support the double-float floating-point data type.

The arguments of this AST must evaluate to double-float values. The value is a Boolean indicating whether the first argument is strictly less than the second argument.

4.4.24 `double-float-not-greater-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementation that support the double-float floating-point data type.

The arguments of this AST must evaluate to double-float values. The value is a Boolean indicating whether the first argument is less than or equal to the second argument.

4.4.25 `double-float-greater-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementation that support the double-float floating-point data type.

The arguments of this AST must evaluate to double-float values. The value is a Boolean indicating whether the first argument is strictly greater than the second argument.

4.4.26 `double-float-not-less-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementation that support the double-float floating-point data type.

The arguments of this AST must evaluate to double-float values. The value is a Boolean indicating whether the first argument is greater than or equal to the

second argument.

4.4.27 `double-float-equal-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementation that support the double-float floating-point data type.

The arguments of this AST must evaluate to double-float values. The value is a Boolean indicating whether the first argument is equal to the second argument.

4.4.28 `long-float-add-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementation that support the long-float floating-point data type.

The arguments of this AST must evaluate to long-float values. The value of this AST is the long-float value representing the sum of the two arguments.

4.4.29 `long-float-sub-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementation that support the long-float floating-point data type.

The arguments of this AST must evaluate to long-float values. The value of this AST is the long-float value representing the difference of the two arguments.

4.4.30 `long-float-mul-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementation that support the long-float floating-point data type.

The arguments of this AST must evaluate to long-float values. The value of this AST is the long-float value representing the product of the two arguments.

4.4.31 `long-float-div-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementation that support the long-float floating-point data type.

The arguments of this AST must evaluate to long-float values. The value of this AST is the long-float value representing the quotient of the two arguments.

4.4.32 `long-float-less-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementation that support the long-float floating-point data type.

The arguments of this AST must evaluate to long-float values. The value is a Boolean indicating whether the first argument is strictly less than the second argument.

4.4.33 `long-float-not-greater-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementation that support the long-float floating-point data type.

The arguments of this AST must evaluate to long-float values. The value is a Boolean indicating whether the first argument is less than or equal to the second argument.

4.4.34 `long-float-greater-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementation that support the long-float floating-point data type.

The arguments of this AST must evaluate to long-float values. The value is a Boolean indicating whether the first argument is strictly greater than the second argument.

4.4.35 `long-float-not-less-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementation that support the long-float floating-point data type.

The arguments of this AST must evaluate to long-float values. The value is a Boolean indicating whether the first argument is greater than or equal to the

second argument.

4.4.36 `long-float-equal-ast`

Initarg	Reader	Description
`:argument1-ast`	`argument1-ast`	The first argument.
`:argument2-ast`	`argument2-ast`	The second argument.

This AST can be used by implementation that support the long-float floating-point data type.

The arguments of this AST must evaluate to long-float values. The value is a Boolean indicating whether the first argument is equal to the second argument.

Chapter 5

Source tracking

We implement source tracking by having the function for converting source named `cleavir-generate-ast:generate-ast` accept a *concrete syntax tree* (or *CST* for short) as its first argument, rather than just a source expression.

A CST has the exact same structure as the corresponding tree representing the source expression, except that some of the nodes in the tree have been augmented with information on source location.

We can describe the structure of the CST for some expression E by means of a transformation C operating on E as follows:

- $C[E] = CST[S, E, ()]$ if E is an atom.
- $C[E] = CST[S, E, (C[e_1]\ C[e_2]\ \cdots\ C[e_n])]$ if E is the list $(e_1\ e_2\ \cdots\ e_n)$.
- $C[E] = CST[S, E, (C[e_1]\ C[e_2]\ \cdots\ C[e_{n-1}]\ .\ C[e_n])]$ if E is the list $(e_1\ e_2\ \cdots\ e_{n-1}\ .\ e_n)$ where e_n is an atom other than `nil`.

In these equations, CST is the constructor for CSTs, S is the source information relating to E. As we can see, the constructor for CSTs takes three arguments: the source information for the expression, the expression itself and the list of transformed *children* of E. If E is a proper list, then the elements of that list are considered the children of E. If E is a dotted list, then the elements of the list and the *atomic tail* of the list are considered the children of E.

An implementation has basically two options for creating a CST:

- It can provide a special version of `read` that returns a CST rather than the raw expression.
- It can take the raw expression returned by an ordinary `read` together with source information about the components of that expression and create a CST.

For example, an implementation that maintains an `eq` hash table mapping every `cons` of the raw expression to some source location (but that does not provide source information for atoms) can create a CST using the following technique:

```
(defun make-cst (expression table)
  (make-instance 'cst
    :expression expression
    :source-info (gethash expression table)
    :children
    (if (atom expression)
        nil
        (loop with result = '()
              for remaining = expression then (rest remaining)
              until (atom remaining)
              do (push (make-cst (first remaining) table) result)
              finally (return (append (nreverse result)
                                      (if (null remaining)
                                          nil
                                          (make-cst remaining table))))))))
```

Chapter 6

Intermediate representation

6.1 Instruction graph

The compiler translates an abstract syntax tree into a *graph* of *instructions*.

This graph is a variation on a *control flow graph*. As such, it has a unique instruction called the *initial instruction* which is the instruction where the execution of the program starts.

In general an instruction can have zero, one, or more *successors*, and zero, one, or more *predecessors*.

An instruction can also have zero, one, or several *inputs* and zero, one, or more *outputs*. An input may be an *constant* value or a *lexical variable*.[1] An output may only be a lexical variable.

The execution of an instruction consists of generating the outputs as a function of the inputs, and also of choosing a successor based on the inputs.

The execution of a program consists of starting execution at the *initial instruction* and executing a sequence of instructions where an instruction in the sequence is the successor chosen during the execution of the preceding instruction in the sequence.

An instruction graph is said to be *well formed* if and only if the restrictions on each of the instructions are respected, as defined in Section 6.3.

[1] The only exception to this rule is that an `enclose-instruction` has an `enter-instruction` as its input.

It is possible for a well-formed instruction graph to contain instructions that are not *reachable* in that there is no execution path from the initial instruction to such an instruction. This situation can arise as a result of certain optimizations that determine that a particular successor arc of some instruction can never be chosen, and so removes that arc. In certain cases, the instruction at the head of that arc may then not be reachable from the initial instruction.

Because of the existence of `enclose-instructions`, the concept of reachability is actually a bit more complicated than what was hinted in the previous paragraph. More formally, an instruction is reachable if and only if:

- it is an `enter-instruction`, or
- it has a reachable predecessor.

By stating that every `enter-instruction` is reachable a priori, we implicitly assume that the closure generated as the output of every `enclose-instruction` is actually *used*. Whether this is the case or not is of course *undecidable*, but the translation from an abstract syntax tree to an instruction graph eliminates some of the cases where the closure may not be used, in that it does not generate output for anonymous functions that are evaluated in a context where the resulting closure is obviously not needed.

An instruction graph program is said to be *reduced* if and only if it is *well formed* and every instruction is *reachable*.

6.2 Levels of detail

From its initial creation, the instruction graph is transformed in various ways. A transformation may add or delete instructions, alter the predecessor/successor relationship between instructions, or add or delete inputs or outputs.

Furthermore, some transformations may add new *types* of instructions that were not present before, and some transformations may remove all instructions of a particular type.

Globally speaking, however, the instruction graph comes in two major varieties defined by what types of instructions it contains. These two varieties are called *High-level Intermediate Representation* (or HIR) and *Medium-level Intermediate Representation* (or MIR) respectively.

6.2.1 High-level Intermediate Representation

At the HIR level, no address calculations are exposed. All variables contain Common Lisp objects, except that some variables may contain *unboxed* or *raw* versions of such objects, in particular integers, floating-point numbers, and characters.

Argument parsing is not exposed, and is instead hidden in a single `enter-instruction`. Similarly, the details of how functions are called and how values are returned is hidden. A single instruction is used to access the fields of a `cons` cell, the element of an array, and the slot of a standard instance.

6.2.2 Initial instruction graph

The initial instruction graph is created from an abstract syntax tree as described in Chapter 4. The following instruction classes can appear in this initial graph:

`enter-instruction`	Section 6.3.1
`nop-instruction`	Section 6.3.2
`assignment-instruction`	Section 6.3.3
`funcall-instruction`	Section 6.3.4
`return-instruction`	Section 6.3.6
`enclose-instruction`	Section 6.3.7
`eq-instruction`	Section 6.3.8
`car-instruction`	Section 6.3.9
`cdr-instruction`	Section 6.3.10
`rplaca-instruction`	Section 6.3.11
`rplacd-instruction`	Section 6.3.12
`slot-readinstruction`	Section 6.3.13
`slot-write-instruction`	Section 6.3.14

6.2.3 Level 1

The transformation from the previous level to this level consists of a *closure analysis*. Lexical locations are divided into two sub-categories, namely *simple locations* and *shared locations*.

A simple location is a lexical location that is assigned and used within a single *function*, which makes it possible to allocate it in a register or in the stack frame belonging to that function.

A shared location, on the other hand, is a location that is assigned and/or used in more than one function. Such a variable can not be allocated in a register or in the stack frame, and must instead be allocated in the *static runtime environment*.

At this level, we also introduce explicit instructions for *non-local control transfers*, resulting from a `return-from` or `go` special form where the origin is in one function and the destination is in a different function.

6.2.4 Level 2

The distinction between simple locations and shared locations is necessary because only simple locations are subject to conversion to *static single assignment* (SSA) form, which is what this level is about.

A priori, every simple location is translated this way, but implementations can leave out some or all simple locations from this conversion. However, some optimizations apply only to simple locations.

The representation at this level is used for *type inference*. The result of this operation is that some parts of the instruction graph may be removed.

Since parts of the instruction graph may be removed, it could be the case that variables that were previously considered shared no longer are. Therefore, additional variables may be converted to SSA form here.

6.2.5 Level 3

This is the first level where real implementation-specific and backend-specific details is introduced. The instructions to access fields of a `cons` cell, elements of an array, or slots of a standard object, are expanded to expose tagging and address calculations.

This is the level where most traditional optimizations are accomplished, such as *value numbering*, *redundancy elimination*, *common subexpression elimination*, etc.

6.2.6 Level 4

At this level, *registers* and *stack locations* are introduced according to the conventions for the implementation and the backend.

This is the level where *register allocation* is accomplished.

6.3 Definition of instructions

6.3.1 Instruction `enter-instruction`

Number of inputs	0
Number of outputs	any
Number of predecessors	0
Number of successors	1

An instruction of this type is either the initial instruction of an instruction graph, or an input to an `enclose-instruction`. The outputs of this instruction corresponds to the parameters the function as follows:

- There is an output for every required parameter.
- For each optional parameter and for each keyword parameter, there are two outputs, one for the parameter itself and one for the `supplied-p` parameter.

The execution of this instruction involves parsing the arguments given to the function, and setting the outputs accordingly.

This instruction has an accessor named `lambda-list`. The lambda list of the `enter-instruction` resembles an ordinary lambda list in that it has a number of required parameters, possibly a `&rest` parameter and some `&optional` and `&key` parameters. It differs from an ordinary lambda list in the following way:

- Each required parameter is a `lexical-location` which is also one of the outputs of the instruction.
- If the `&rest` lambda list keyword is present, it is followed by a `lexical-location` which is also one of the outputs of the instruction.
- If the `&optional` lambda list keyword is present, it is followed by any number of two-element lists. Each element of each such lists is a `lexical-location` which is also one of the outputs of the instruction. The first element represents the argument if it was given and the second element represents a `supplied-p` argument containing either `nil` or `t`.
- If the `&key` lambda list keyword is present, it is followed by any number of three-element lists. The first element of each such list is a *symbol* (typically a symbol in the `keyword` package) used to recognize whether the corresponding argument has been given. The second element and the third element of each list is a `lexical-location` which is also one of the outputs of the instruction. The

second element represents the argument if it was given and the third element represents a `supplied-p` argument containing either `nil` or `t`.

Notice that the outputs are lexical locations independently of whether the parameters to the function are special variables. The MIR instructions following the `enter-instruction` are responsible for binding special variables and assigning default values to unsupplied `&optional` and `&key` parameters.

Figure 6.1 shows the Graphviz illustration of the `enter-instruction`

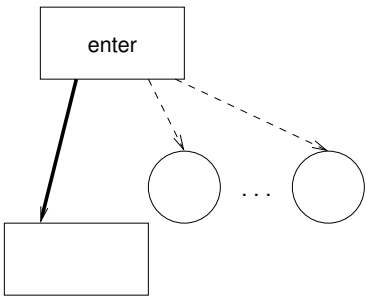

Figure 6.1: `enter-instruction`.

6.3.2 Instruction `nop-instruction`

Number of inputs	0
Number of outputs	0
Number of predecessors	any
Number of successors	1

Executing this instruction has no effect. This instruction may be introduced by certain transformations as a replacement for existing instructions. This way, these transformations do not have to be concerned with complex updates of the graph. Instead a dedicated transformation for removing all `nop-instruction`s is provided.

Figure 6.2 shows the Graphviz illustration of the `nop-instruction`

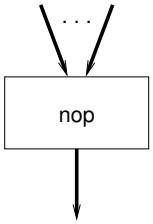

Figure 6.2: `nop-instruction`.

6.3.3 Instruction `assignment-instruction`

Number of inputs	1
Number of outputs	1
Number of predecessors	any
Number of successors	1

The execution of this instruction results in the input being copied to the output without any modification.

Figure 6.3 shows the Graphviz illustration of the `assignment-instruction`

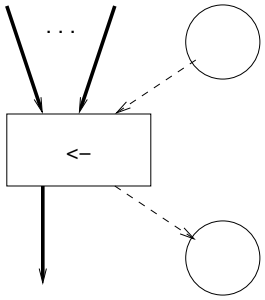

Figure 6.3: `assignment-instruction`.

6.3.4 Instruction `funcall-instruction`

Number of inputs	any
Number of outputs	any
Number of predecessors	any
Number of successors	1

The first input of this instruction corresponds to the function being called. The remaining inputs correspond to the arguments to be passed to the callee.

The outputs of this instruction correspond to the values that the caller requests from the callee. The number of outputs can therefore be different from the actual number of values produced by the caller.

Figure 6.4 shows the Graphviz illustration of the `funcall-instruction`

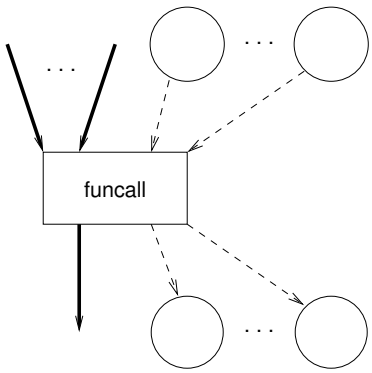

Figure 6.4: `funcall-instruction`.

6.3.5 Instruction `tailcall-instruction`

Number of inputs	any
Number of outputs	0
Number of predecessors	any
Number of successors	0

The first input of this instruction corresponds to the function being called. The remaining inputs correspond to the arguments to be passed to the callee.

Figure 6.5 shows the Graphviz illustration of the `tailcall-instruction`

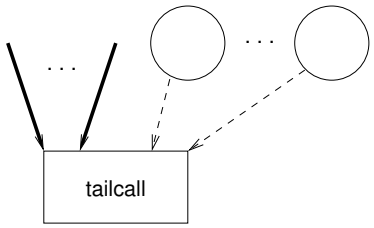

Figure 6.5: `tailcall-instruction`.

6.3.6 Instruction `return-instruction`

Number of inputs	any
Number of outputs	0
Number of predecessors	any
Number of successors	0

The inputs of this instruction correspond to the values transmitted to the caller

It terminates execution of the current function and returns to the caller.

Figure 6.6 shows the Graphviz illustration of the `return-instruction`

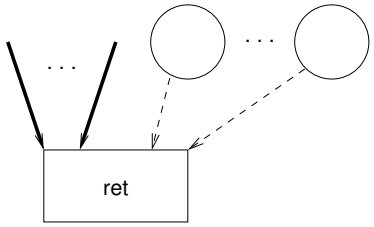

Figure 6.6: `return-instruction`.

6.3.7 Instruction `enclose-instruction`

Number of inputs	1
Number of outputs	1
Number of predecessors	any
Number of successors	1

The input of this instruction is different from those of other instructions because it is another instruction, namely an `enter-instruction`.

The instruction takes the instruction graph of the input and creates a *closure*. The closure contains the current lexical runtime environment and the code resulting from the input instruction graph.

Figure 6.7 shows the Graphviz illustration of the `enclose-instruction`

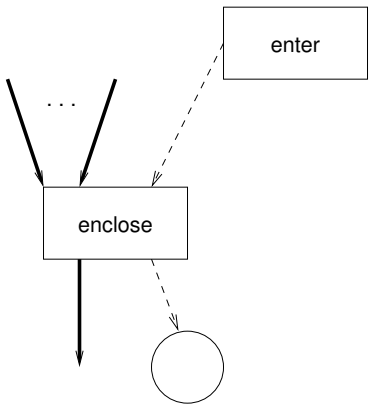

Figure 6.7: `enclose-instruction`.

6.3.8 Instruction `eq-instruction`

Number of inputs	2
Number of outputs	0
Number of predecessors	any
Number of successors	2

The input of this instruction is a generalized Boolean. The first successor is chosen if the value of the input is *true* and the second successor is chosen of the value of the input is *false*.

Figure 6.8 shows the Graphviz illustration of the `eq-instruction`

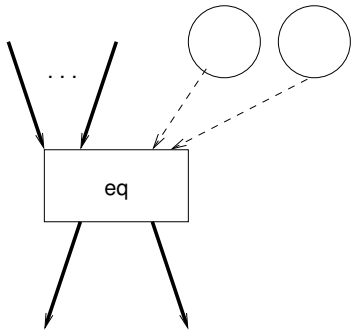

Figure 6.8: `eq-instruction`.

6.3.9 Instruction `car-instruction`

Number of inputs	1
Number of outputs	1
Number of predecessors	any
Number of successors	1

The input to this instruction must be a `cons` cell. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The output is the contents of the `car` of the `cons` cell.

Figure 6.9 shows the Graphviz illustration of the `car-instruction`

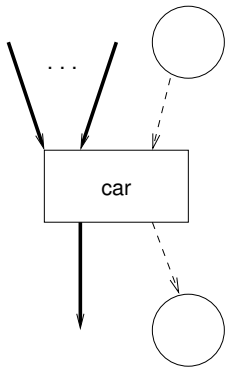

Figure 6.9: `car-instruction`.

6.3.10 Instruction `cdr-instruction`

Number of inputs	1
Number of outputs	1
Number of predecessors	any
Number of successors	1

The input to this instruction must be a `cons` cell. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The output is the contents of the `cdr` of the `cons` cell.

Figure 6.10 shows the Graphviz illustration of the `cdr-instruction`

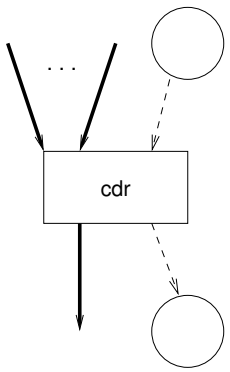

Figure 6.10: `cdr-instruction`.

6.3.11 Instruction `rplaca-instruction`

Number of inputs	2
Number of outputs	0
Number of predecessors	any
Number of successors	1

The first input to this instruction must be a `cons` cell. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The second input can be any object. The effect of the instruction is to replace the contents of the `car` of the `cons` cell by the second input.

Figure 6.11 shows the Graphviz illustration of the `rplaca-instruction`

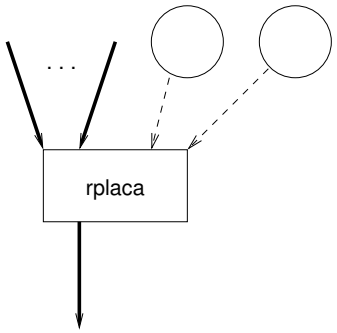

Figure 6.11: `rplaca-instruction`.

6.3.12 Instruction `rplacd-instruction`

Number of inputs	2
Number of outputs	0
Number of predecessors	any
Number of successors	1

The first input to this instruction must be a `cons` cell. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The second input can be any object. The effect of the instruction is to replace the contents of the `cdr` of the `cons` cell by the second input.

Figure 6.12 shows the Graphviz illustration of the `rplacd-instruction`

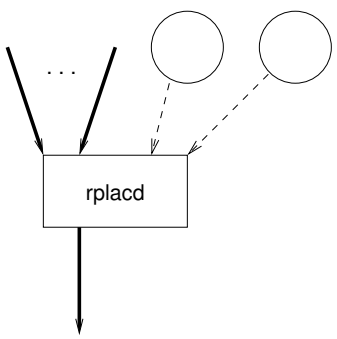

Figure 6.12: `rplacd-instruction`.

6.3.13 Instruction `slot-read-instruction`

Number of inputs	2
Number of outputs	1
Number of predecessors	any
Number of successors	1

The first input to this instruction must be a *standard object*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The second input is a non-negative fixnum indicating the *index* of the slot in the instance. The output is the contents of the corresponding slot.

Figure 6.13 shows the Graphviz illustration of the `slot-read-instruction`

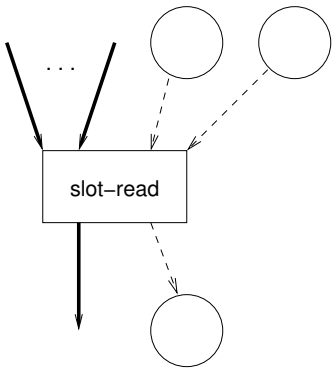

Figure 6.13: `slot-read-instruction`.

6.3.14 Instruction `slot-write-instruction`

Number of inputs	3
Number of outputs	0
Number of predecessors	any
Number of successors	1

The first input to this instruction must be a *standard object*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The second input is a fixnum indicating the *index* of the slot in the instance. The third input can be any object. The effect of the instruction is to replace the contents of the corresponding slot by the third input.

Figure 6.14 shows the Graphviz illustration of the `slot-write-instruction`

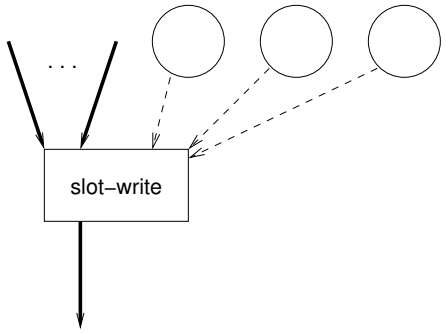

Figure 6.14: `slot-write-instruction`.

6.3.15 Instruction `aref-instruction`

Number of inputs	2
Number of outputs	1
Number of predecessors	any
Number of successors	1

The first input to this instruction must be a *general array*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The second input is a non-negative fixnum indicating the *row-major index* of an element of the array. The output is the contents of the corresponding element of the array.

Figure 6.15 shows the Graphviz illustration of the `aref-instruction`

6.3.16 Instruction `short-float-aref-instruction`

Number of inputs	2
Number of outputs	1
Number of predecessors	any
Number of successors	1

This instruction can be used by implementations that support arrays specialized to `short-float`.

The first input to this instruction must be an *array* specialized to `short-float`. If the

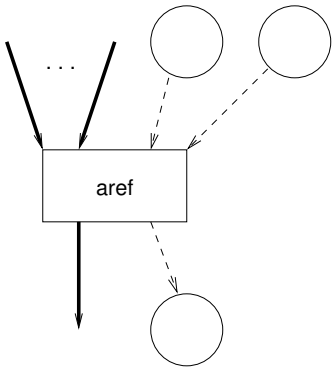

Figure 6.15: `aref-instruction`.

MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The second input is a non-negative fixnum indicating the *row-major index* of an element of the array. The output is the contents of the corresponding element of the array. The output is an unboxed `short-float` number.

Figure 6.16 shows the Graphviz illustration of the `short-float-aref-instruction`

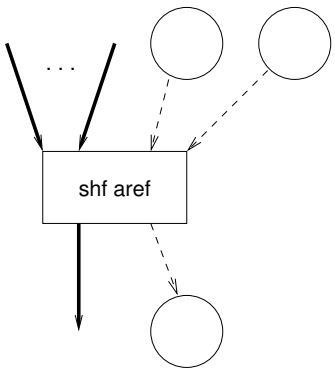

Figure 6.16: `short-float-aref-instruction`.

6.3.17 Instruction `single-float-aref-instruction`

Number of inputs	2
Number of outputs	1
Number of predecessors	any
Number of successors	1

This instruction can be used by implementations that support arrays specialized to `single-float`.

The first input to this instruction must be an *array* specialized to `single-float`. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The second input is a non-negative fixnum indicating the *row-major index* of an element of the array. The output is the contents of the corresponding element of the array. The output is an unboxed `single-float` number.

Figure 6.17 shows the Graphviz illustration of the `single-float-aref-instruction`

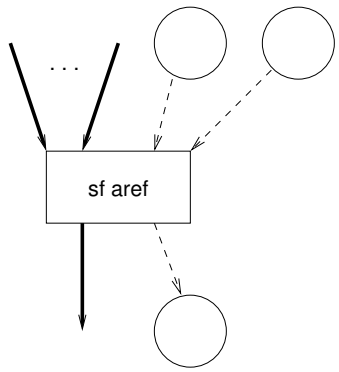

Figure 6.17: `single-float-aref-instruction`.

6.3.18 Instruction `double-float-aref-instruction`

Number of inputs	2
Number of outputs	1
Number of predecessors	any
Number of successors	1

This instruction can be used by implementations that support arrays specialized to

`double-float`.

The first input to this instruction must be an *array* specialized to `double-float`. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The second input is a non-negative fixnum indicating the *row-major index* of an element of the array. The output is the contents of the corresponding element of the array. The output is an unboxed `double-float` number.

Figure 6.18 shows the Graphviz illustration of the `double-float-aref-instruction`

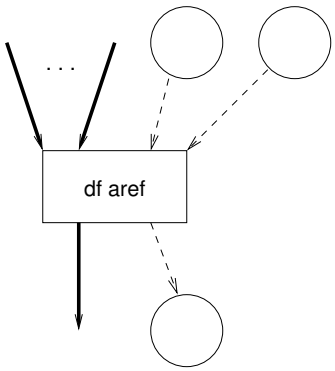

Figure 6.18: `double-float-aref-instruction`.

6.3.19 Instruction `long-float-aref-instruction`

Number of inputs	2
Number of outputs	1
Number of predecessors	any
Number of successors	1

This instruction can be used by implementations that support arrays specialized to `long-float`.

The first input to this instruction must be an *array* specialized to `long-float`. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The second input is a non-negative fixnum indicating the *row-major index* of an element of the array. The output is the contents of the corresponding element of the array. The output is an unboxed `long-float` number.

Figure 6.19 shows the Graphviz illustration of the `long-float-aref-instruction`

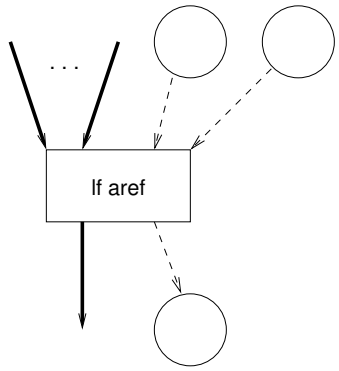

Figure 6.19: `long-float-aref-instruction`.

6.3.20 Instruction `bit-aref-instruction`

Number of inputs	2
Number of outputs	1
Number of predecessors	any
Number of successors	1

This instruction is used by for accessing an element of an array specialized to `bit`.

The first input to this instruction must be an *array* specialized to `bit`. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The second input is a non-negative fixnum indicating the *row-major index* of an element of the array. The output is the contents of the corresponding element of the array. The output is an unboxed `bit`.

Figure 6.20 shows the Graphviz illustration of the `bit-aref-instruction`

6.3.21 Instruction `aset-instruction`

Number of inputs	3
Number of outputs	0
Number of predecessors	any
Number of successors	1

The first input to this instruction must be a *general array*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The

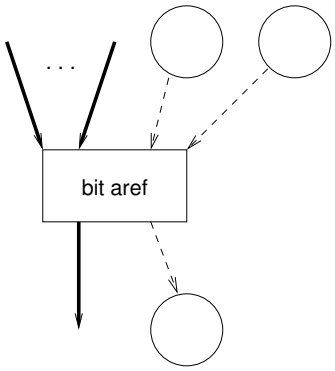

Figure 6.20: `bit-aref-instruction`.

second input is a fixnum indicating the *row-major index* of an element of the array. The third input can be any object. The effect of the instruction is to replace the contents of the corresponding element by the third input.

Figure 6.21 shows the Graphviz illustration of the `aset-instruction`

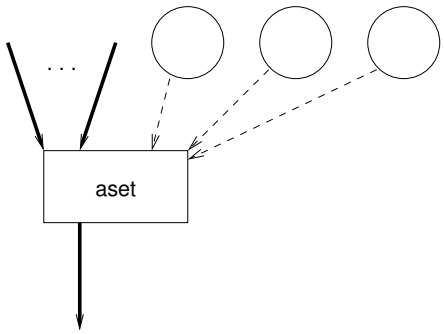

Figure 6.21: `aset-instruction`.

6.3.22 Instruction `short-float-aset-instruction`

Number of inputs	3
Number of outputs	0
Number of predecessors	any
Number of successors	1

This instruction can be used by implementations that support arrays specialized to `short-float`.

The first input to this instruction must be an *array* specialized to `short-float`. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The second input is a non-negative fixnum indicating the *row-major index* of an element of the array. The third input is an unboxed `short-float` number.

The effect of the instruction is to replace the contents of the corresponding element by the third input.

Figure 6.22 shows the Graphviz illustration of the `short-float-aset-instruction`

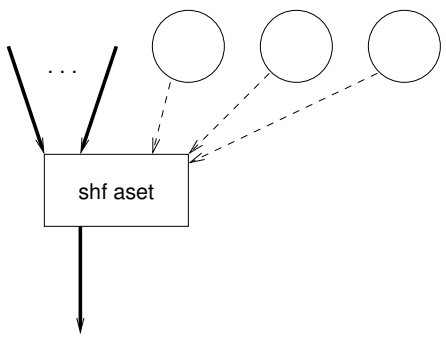

Figure 6.22: `short-float-aset-instruction`.

6.3.23 Instruction `single-float-aset-instruction`

Number of inputs	3
Number of outputs	0
Number of predecessors	any
Number of successors	1

This instruction can be used by implementations that support arrays specialized to

`single-float`.

The first input to this instruction must be an *array* specialized to `single-float`. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The second input is a non-negative fixnum indicating the *row-major index* of an element of the array. The third input is an unboxed `single-float` number.

The effect of the instruction is to replace the contents of the corresponding element by the third input.

Figure 6.23 shows the Graphviz illustration of the `single-float-aset-instruction`

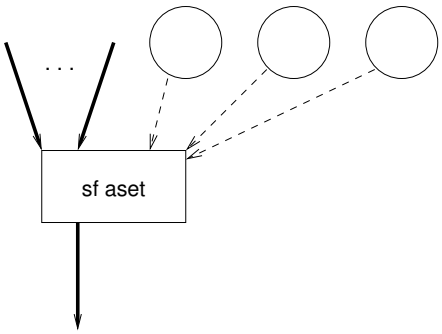

Figure 6.23: `single-float-aset-instruction`.

6.3.24 Instruction `double-float-aset-instruction`

Number of inputs	3
Number of outputs	0
Number of predecessors	any
Number of successors	1

This instruction can be used by implementations that support arrays specialized to `double-float`.

The first input to this instruction must be an *array* specialized to `double-float`. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The second input is a non-negative fixnum indicating the *row-major index* of an element of the array. The third input is an unboxed `double-float` number.

The effect of the instruction is to replace the contents of the corresponding element

by the third input.

Figure 6.24 shows the Graphviz illustration of the `double-float-aset-instruction`

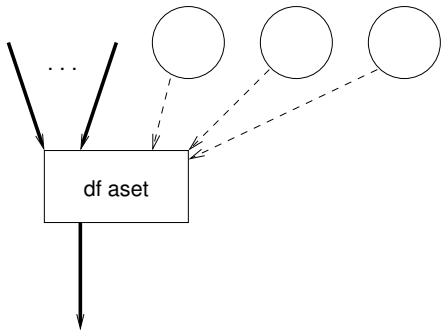

Figure 6.24: `double-float-aset-instruction`.

6.3.25 Instruction `long-float-aset-instruction`

Number of inputs	3
Number of outputs	0
Number of predecessors	any
Number of successors	1

This instruction can be used by implementations that support arrays specialized to `long-float`.

The first input to this instruction must be an *array* specialized to `long-float`. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The second input is a non-negative fixnum indicating the *row-major index* of an element of the array. The third input is an unboxed `long-float` number.

The effect of the instruction is to replace the contents of the corresponding element by the third input.

Figure 6.25 shows the Graphviz illustration of the `long-float-aset-instruction`

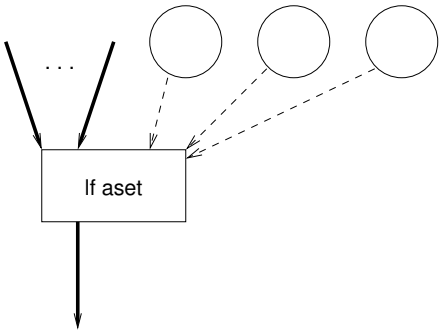

Figure 6.25: `long-float-aset-instruction`.

6.3.26 Instruction `bit-aset-instruction`

Number of inputs	3
Number of outputs	0
Number of predecessors	any
Number of successors	1

This instruction is used by for setting an element of an array specialized to `bit`.

The first input to this instruction must be an *array* specialized to `bit`. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The second input is a non-negative fixnum indicating the *row-major index* of an element of the array. The third input is an unboxed `bit`.

The effect of the instruction is to replace the contents of the corresponding element by the third input.

Figure 6.26 shows the Graphviz illustration of the `bit-aset-instruction`

6.3.27 Instruction `fixnum-add-instruction`

Number of inputs	2
Number of outputs	1
Number of predecessors	any
Number of successors	2

The inputs to this instruction must be of type `fixnum`. If the MIR program is *safe*,

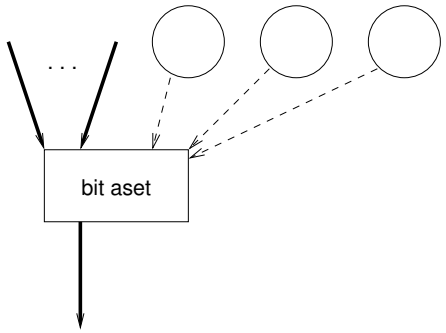

Figure 6.26: `bit-aset-instruction`.

then control must reach this instruction only if this restriction is verified. The output is a `fixnum`.

The output is the sum of the two inputs.

If the second successor is chosen, then in order to obtain the sum of the two inputs, the following adjustment has to be made:

- If the output is negative, then 2^n must be added to the output, where n is the number of bits used to represent a fixnum.
- If the output is positive, then 2^n must be subtracted from the output, where n is the number of bits used to represent a fixnum.

Figure 6.27 shows the Graphviz illustration of the `fixnum-add-instruction`

6.3.28 Instruction `fixnum-sub-instruction`

Number of inputs	2
Number of outputs	1
Number of predecessors	any
Number of successors	2

The inputs to this instruction must be of type `fixnum`. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The output is a `fixnum`.

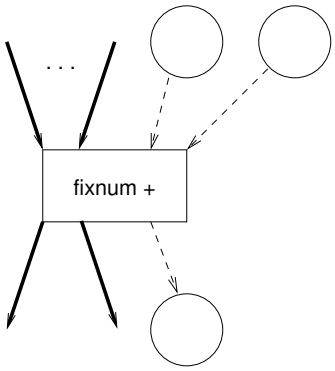

Figure 6.27: `fixnum-add-instruction`.

The output is the difference between the two inputs.

If the second successor is chosen, then in order to obtain the difference between the two inputs, the following adjustment has to be made:

- If the output is negative, then 2^n must be added to the output, where n is the number of bits used to represent a fixnum.
- If the output is positive, then 2^n must be subtracted from the output, where n is the number of bits used to represent a fixnum.

Figure 6.28 shows the Graphviz illustration of the `fixnum-sub-instruction`

6.3.29 Instruction `fixnum-<-instruction`

Number of inputs	2
Number of outputs	0
Number of predecessors	any
Number of successors	2

The inputs to this instruction must be of type `fixnum`. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified.

The first successor is chosen if and only if the first input is strictly less than the second input.

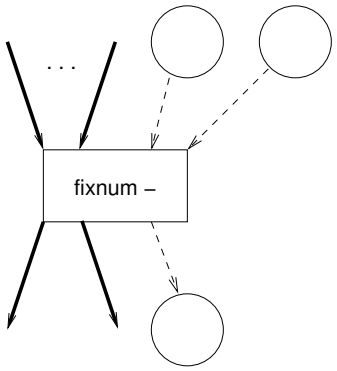

Figure 6.28: `fixnum-sub-instruction`.

Figure 6.29 shows the Graphviz illustration of the `fixnum-<-instruction`

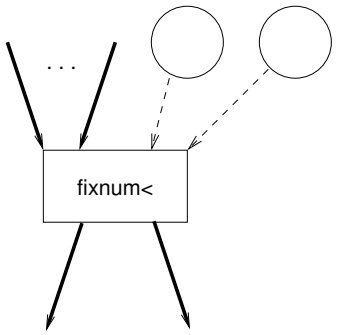

Figure 6.29: `fixnum-<-instruction`.

6.3.30 Instruction `fixnum-<=-instruction`

Number of inputs	2
Number of outputs	0
Number of predecessors	any
Number of successors	2

The inputs to this instruction must be of type `fixnum`. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified.

The first successor is chosen if and only if the first input is less than or equal to the second input.

Figure 6.30 shows the Graphviz illustration of the `fixnum-<=-instruction`

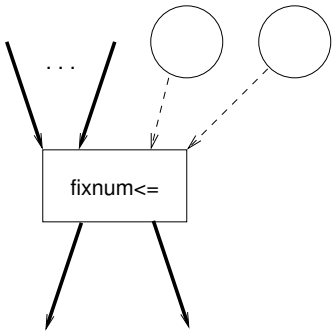

Figure 6.30: `fixnum-<=-instruction`.

6.3.31 Instruction `fixnum-=-instruction`

Number of inputs	2
Number of outputs	0
Number of predecessors	any
Number of successors	2

The inputs to this instruction must be of type `fixnum`. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified.

The first successor is chosen if and only if the first input is equal to the second input.

Figure 6.31 shows the Graphviz illustration of the `fixnum-=-instruction`

6.3.32 Instruction `phi-instruction`

Number of inputs	> 1
Number of outputs	1
Number of predecessors	any
Number of successors	1

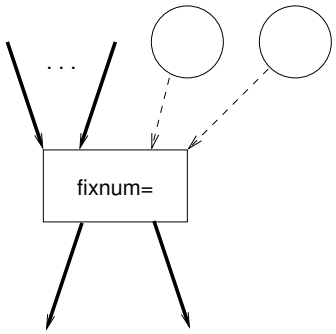

Figure 6.31: `fixnum-=-instruction`.

This instruction is used for variables respecting the *static single assignment* (SSA) property.

One of the following must hold for any `phi-instruction` A:

- The number of predecessors of A is equal to the number of inputs to A, or
- A has a single predecessor B, and B is also a `phi-instruction` with the same number of inputs as A.

In other words, `phi-instruction`s occur in *clusters* with the same number of inputs n. The first instruction of such a cluster has n predecessors, and the others have a single predecessor.

Special care must be taken when the graph is modified so that this restriction is respected.

Figure 6.32 shows the Graphviz illustration of the `phi-instruction`

Some transformations may be easier to accomplish if there is a single `phi-instruction` as opposed to a cluster of several such instructions. We may propose a *cluster/uncluster* transformation on the entire program so that both representations are accessible.

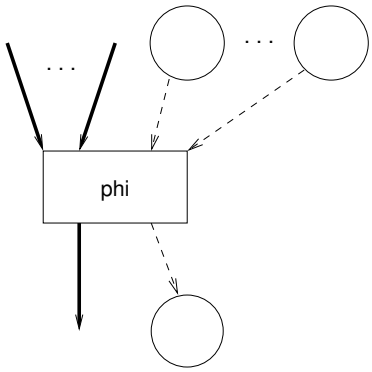

Figure 6.32: `phi-instruction`.

6.3.33 Instruction `typeq-instruction`

Number of inputs	2
Number of outputs	0
Number of predecessors	any
Number of successors	2

The first input of this instruction is an object of which the type is be tested. The second input is a *type specifier*. Initially, the second input is a `constant-input`, but that might change as a result of transformations of the graph.

The first successor is chosen if and only if the first input is of the type denoted by the second input.

This instruction is used in a variety of situations:

- When the source code contains a call to `typep` with a constant type argument, a compiler macro might transform the call to a special form that generates this instruction.
- Some implementations might generate this instruction from the special operator `the`, where the second successor contains a call to `error`.
- Since type declarations can be seen as implicit use of the special operator `the`, this instruction can be used for type declarations as well.
- The macro `check-type` might result in this instruction being generated, in

which case the second successor would contain instructions to signal a correctable error.

Because the second input may not be a `constant-input`, the type specifier can also be retrieved using the slot reader `value-type`.

Figure 6.33 shows the Graphviz illustration of the `typeq-instruction`

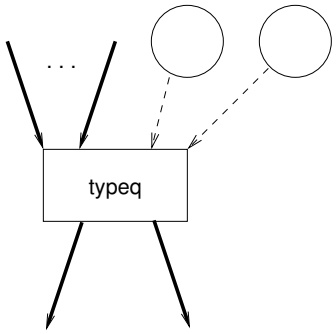

Figure 6.33: `typeq-instruction`.

6.3.34 Instruction `sref-instruction`

Number of inputs	1
Number of outputs	1
Number of predecessors	any
Number of successors	1

The input to this instruction must be a constant symbol naming a special variable. The output is the value of that special variable.

Figure 6.34 shows the Graphviz illustration of the `sref-instruction`

6.3.35 Instruction `sset-instruction`

Number of inputs	2
Number of outputs	0
Number of predecessors	any
Number of successors	1

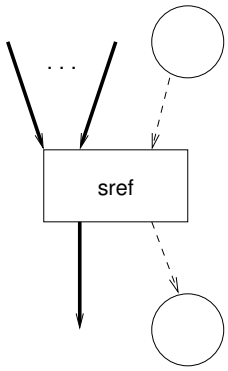

Figure 6.34: `sref-instruction`.

The first input to this instruction must be a constant symbol naming a special variable. The second input can be any object. The effect of the instruction is to replace the current value of the special variable the second input.

Figure 6.35 shows the Graphviz illustration of the `sset-instruction`

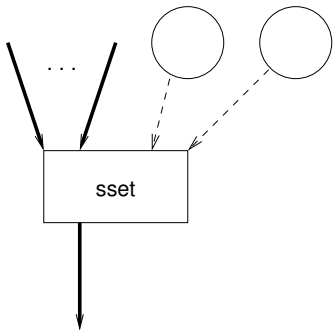

Figure 6.35: `sset-instruction`.

6.3.36 Instruction `sbind-instruction`

Number of inputs	2
Number of outputs	0
Number of predecessors	any
Number of successors	1

The first input to this instruction must be a constant symbol naming a special variable. The second input can be any object. The effect of the instruction is to create a new binding for the special variable, where the initial value is that of the second input.

The binding is automatically destroyed when the current function invocation terminates by executing a `return-instruction`. After an instruction of this type has been executed, control flow must not be able to reach a `tailcall-instruction`.

Figure 6.36 shows the Graphviz illustration of the `sbind-instruction`

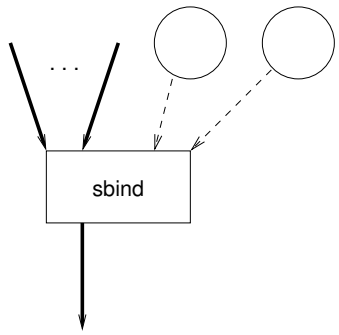

Figure 6.36: `sbind-instruction`.

6.3.37 Instruction `short-float-box-instruction`

Number of inputs	1
Number of outputs	1
Number of predecessors	any
Number of successors	1

The input to this instruction must be an unboxed `short-float` value. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified.

The output is a boxed `short-float` value corresponding to the input.

This instruction can be used by implementations that support the `short-float` data type.

Figure 6.37 shows the Graphviz illustration of the `short-float-box-instruction`

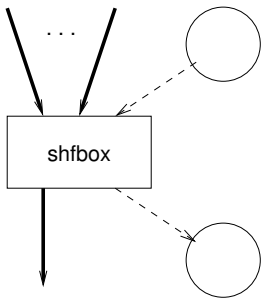

Figure 6.37: `short-float-box-instruction`.

6.3.38 Instruction `short-float-unbox-instruction`

Number of inputs	1
Number of outputs	1
Number of predecessors	any
Number of successors	1

The input to this instruction must be a boxed `short-float` value. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified.

The output is an unboxed `short-float` value corresponding to the input.

This instruction can be used by implementations that support the `short-float` data type.

Figure 6.38 shows the Graphviz illustration of the `short-float-unbox-instruction`

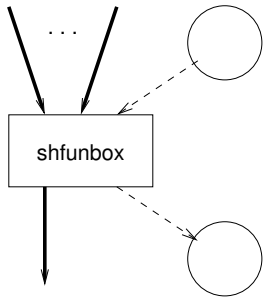

Figure 6.38: `short-float-unbox-instruction`.

6.3.39 Instruction `single-float-box-instruction`

Number of inputs	1
Number of outputs	1
Number of predecessors	any
Number of successors	1

The input to this instruction must be an unboxed `single-float` value. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified.

The output is a boxed `single-float` value corresponding to the input.

Figure 6.39 shows the Graphviz illustration of the `single-float-box-instruction`

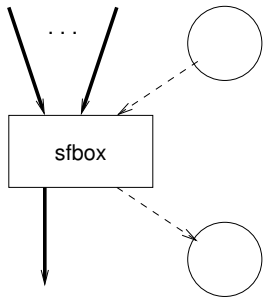

Figure 6.39: `single-float-box-instruction`.

6.3.40 Instruction `single-float-unbox-instruction`

Number of inputs	1
Number of outputs	1
Number of predecessors	any
Number of successors	1

The input to this instruction must be a boxed `single-float` value. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified.

The output is an unboxed `single-float` value corresponding to the input.

Figure 6.40 shows the Graphviz illustration of the `single-float-unbox-instruction`

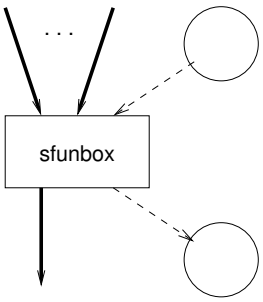

Figure 6.40: `single-float-unbox-instruction`.

6.3.41 Instruction `double-float-box-instruction`

Number of inputs	1
Number of outputs	1
Number of predecessors	any
Number of successors	1

The input to this instruction must be an unboxed `double-float` value. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified.

The output is a boxed `double-float` value corresponding to the input.

This instruction can be used by implementations that support the `double-float` data

type.

Figure 6.41 shows the Graphviz illustration of the `double-float-box-instruction`

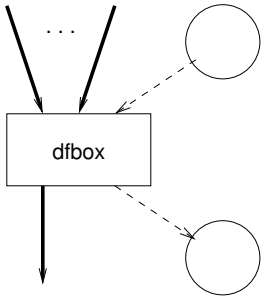

Figure 6.41: `double-float-box-instruction`.

6.3.42 Instruction `double-float-unbox-instruction`

Number of inputs	1
Number of outputs	1
Number of predecessors	any
Number of successors	1

The input to this instruction must be a boxed `double-float` value. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified.

The output is an unboxed `double-float` value corresponding to the input.

This instruction can be used by implementations that support the `double-float` data type.

Figure 6.42 shows the Graphviz illustration of the `double-float-unbox-instruction`

6.3.43 Instruction `long-float-box-instruction`

Number of inputs	1
Number of outputs	1
Number of predecessors	any
Number of successors	1

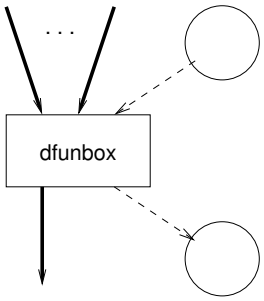

Figure 6.42: `double-float-unbox-instruction`.

The input to this instruction must be an unboxed `long-float` value. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified.

The output is a boxed `long-float` value corresponding to the input.

This instruction can be used by implementations that support the `long-float` data type.

Figure 6.43 shows the Graphviz illustration of the `long-float-box-instruction`

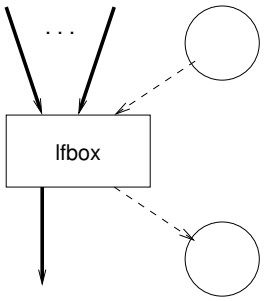

Figure 6.43: `long-float-box-instruction`.

6.3.44 Instruction `long-float-unbox-instruction`

Number of inputs	1
Number of outputs	1
Number of predecessors	any
Number of successors	1

The input to this instruction must be a boxed `long-float` value. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified.

The output is an unboxed `long-float` value corresponding to the input.

This instruction can be used by implementations that support the `long-float` data type.

Figure 6.44 shows the Graphviz illustration of the `long-float-unbox-instruction`

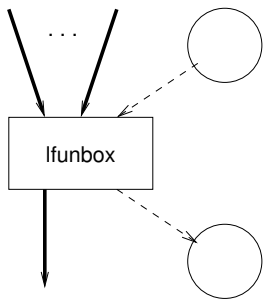

Figure 6.44: `long-float-unbox-instruction`.

6.3.45 Instruction `bit-box-instruction`

Number of inputs	1
Number of outputs	1
Number of predecessors	any
Number of successors	1

The input to this instruction must be an unboxed `bit`. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified.

The output is a boxed `bit` value corresponding to the input, i.e., a fixnum with the value of 0 or 1.

Figure 6.45 shows the Graphviz illustration of the `bit-box-instruction`

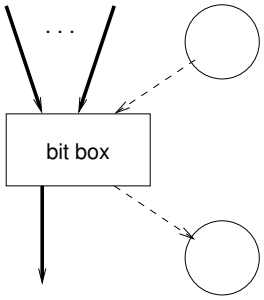

Figure 6.45: `bit-box-instruction`.

6.3.46 Instruction `bit-unbox-instruction`

Number of inputs	1
Number of outputs	1
Number of predecessors	any
Number of successors	1

The input to this instruction must be a boxed `bit` value, i.e., a `fixnum` with a value of either 0 or 1. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified.

The output is an unboxed `bit` value corresponding to the input.

Figure 6.46 shows the Graphviz illustration of the `bit-unbox-instruction`

6.3.47 Instruction `short-float-add-instruction`

Number of inputs	2
Number of outputs	1
Number of predecessors	any
Number of successors	1

The inputs to this instruction must be of type *unboxed short float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The output is also of type unboxed short float.

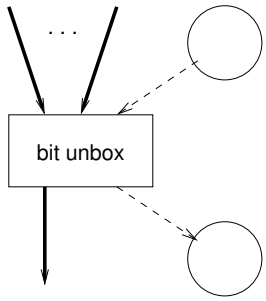

Figure 6.46: `bit-unbox-instruction`.

The output is the sum of the two inputs.

Figure 6.47 shows the Graphviz illustration of the `short-float-add-instruction`

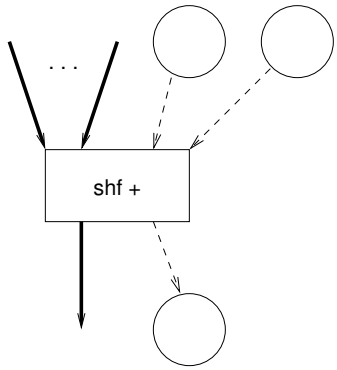

Figure 6.47: `short-float-add-instruction`.

6.3.48 Instruction `short-float-sub-instruction`

Number of inputs	2
Number of outputs	1
Number of predecessors	any
Number of successors	1

The inputs to this instruction must be of type *unboxed short float*. If the MIR program

is *safe*, then control must reach this instruction only if this restriction is verified. The output is also of type unboxed short float.

The output is the difference between the two inputs.

Figure 6.48 shows the Graphviz illustration of the `short-float-sub-instruction`

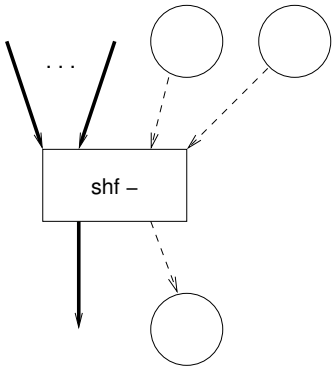

Figure 6.48: `short-float-sub-instruction`.

6.3.49 Instruction `short-float-mul-instruction`

Number of inputs	2
Number of outputs	1
Number of predecessors	any
Number of successors	1

The inputs to this instruction must be of type *unboxed short float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The output is also of type unboxed short float.

The output is the product of the two inputs.

Figure 6.49 shows the Graphviz illustration of the `short-float-mul-instruction`

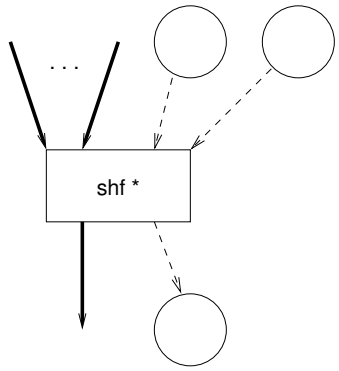

Figure 6.49: `short-float-mul-instruction`.

6.3.50 Instruction `short-float-div-instruction`

Number of inputs	2
Number of outputs	1
Number of predecessors	any
Number of successors	1

The inputs to this instruction must be of type *unboxed short float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The output is also of type unboxed short float.

The output is the quotient of the two inputs.

Figure 6.50 shows the Graphviz illustration of the `short-float-div-instruction`

6.3.51 Instruction `short-float-less-instruction`

Number of inputs	2
Number of outputs	0
Number of predecessors	any
Number of successors	2

The inputs to this instruction must be of type *unboxed short float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified.

The first successor is chosen if the first input is strictly less than the second one;

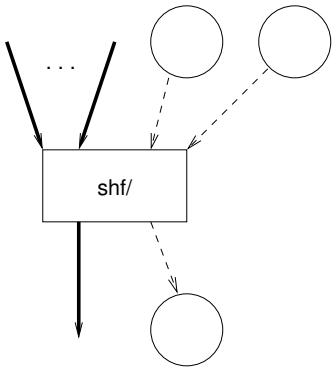

Figure 6.50: `short-float-div-instruction`.

otherwise the second successor is chosen.

Figure 6.51 shows the Graphviz illustration of the `short-float-less-instruction`

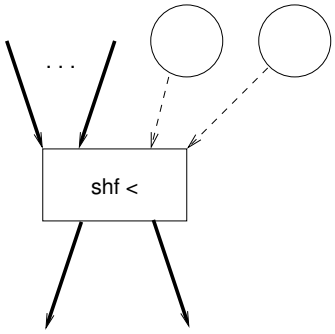

Figure 6.51: `short-float-less-instruction`.

6.3.52 Instruction `short-float-not-greater-instruction`

Number of inputs	2
Number of outputs	0
Number of predecessors	any
Number of successors	2

The inputs to this instruction must be of type *unboxed short float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified.

The first successor is chosen if the first input is less than or equal to the second one; otherwise the second successor is chosen.

Figure 6.52 shows the Graphviz illustration of the `short-float-not-greater-instruction`

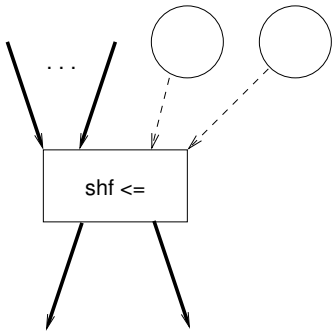

Figure 6.52: `short-float-not-greater-instruction`.

6.3.53 Instruction `short-float-sin-instruction`

Number of inputs	1
Number of outputs	1
Number of predecessors	any
Number of successors	1

The input to this instruction must be of type *unboxed short float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The output is also of type unboxed short float.

The output is the sine of the input.

Figure 6.53 shows the Graphviz illustration of the `short-float-sin-instruction`

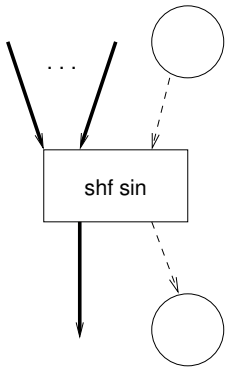

Figure 6.53: `short-float-sin-instruction`.

6.3.54 Instruction `short-float-cos-instruction`

Number of inputs	1
Number of outputs	1
Number of predecessors	any
Number of successors	1

The input to this instruction must be of type *unboxed short float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The output is also of type unboxed short float.

The output is the cosine of the input.

Figure 6.54 shows the Graphviz illustration of the `short-float-cos-instruction`

6.3.55 Instruction `short-float-sqrt-instruction`

Number of inputs	1
Number of outputs	1
Number of predecessors	any
Number of successors	1

The input to this instruction must be of type *unboxed short float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The output is also of type unboxed short float.

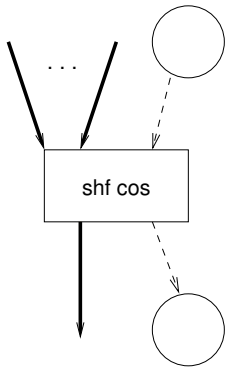

Figure 6.54: `short-float-cos-instruction`.

The output is the square root of the input.

Figure 6.55 shows the Graphviz illustration of the `short-float-sqrt-instruction`

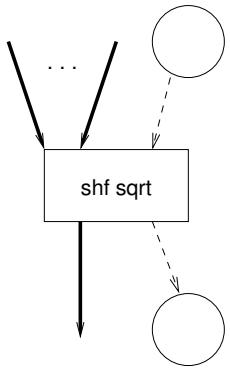

Figure 6.55: `short-float-sqrt-instruction`.

6.3.56 Instruction `single-float-add-instruction`

Number of inputs	2
Number of outputs	1
Number of predecessors	any
Number of successors	1

The inputs to this instruction must be of type *unboxed single float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The output is also of type unboxed single float.

The output is the sum of the two inputs.

Figure 6.56 shows the Graphviz illustration of the `single-float-add-instruction`

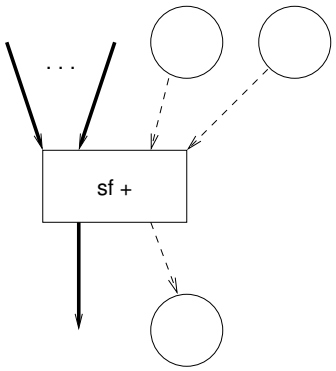

Figure 6.56: `single-float-add-instruction`.

6.3.57 Instruction `single-float-sub-instruction`

Number of inputs	2
Number of outputs	1
Number of predecessors	any
Number of successors	1

The inputs to this instruction must be of type *unboxed single float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The output is also of type unboxed single float.

The output is the difference between the two inputs.

Figure 6.57 shows the Graphviz illustration of the `single-float-sub-instruction`

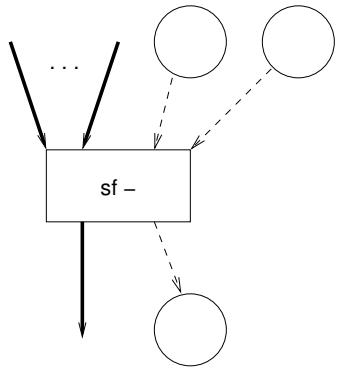

Figure 6.57: `single-float-sub-instruction`.

6.3.58 Instruction `single-float-mul-instruction`

Number of inputs	2
Number of outputs	1
Number of predecessors	any
Number of successors	1

The inputs to this instruction must be of type *unboxed single float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The output is also of type unboxed single float.

The output is the product of the two inputs.

Figure 6.58 shows the Graphviz illustration of the `single-float-mul-instruction`

6.3.59 Instruction `single-float-div-instruction`

Number of inputs	2
Number of outputs	1
Number of predecessors	any
Number of successors	1

The inputs to this instruction must be of type *unboxed single float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The output is also of type unboxed single float.

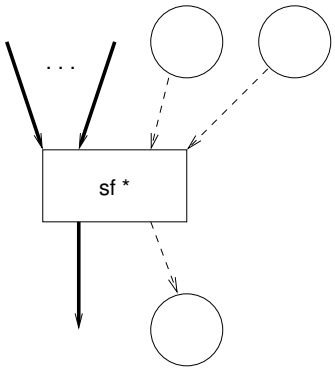

Figure 6.58: `single-float-mul-instruction`.

The output is the quotient of the two inputs.

Figure 6.59 shows the Graphviz illustration of the `single-float-div-instruction`

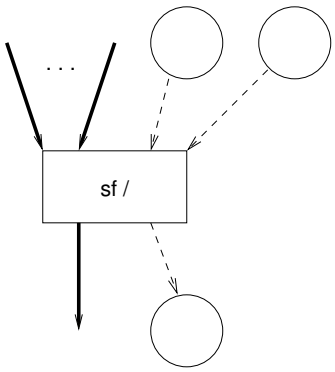

Figure 6.59: `single-float-div-instruction`.

6.3.60 Instruction `single-float-less-instruction`

Number of inputs	2
Number of outputs	0
Number of predecessors	any
Number of successors	2

The inputs to this instruction must be of type *unboxed single float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified.

The first successor is chosen if the first input is strictly less than the second one; otherwise the second successor is chosen.

Figure 6.60 shows the Graphviz illustration of the `single-float-less-instruction`

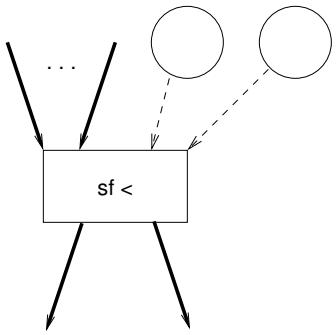

Figure 6.60: `single-float-less-instruction`.

6.3.61 Instruction `single-float-not-greater-instruction`

Number of inputs	2
Number of outputs	0
Number of predecessors	any
Number of successors	2

The inputs to this instruction must be of type *unboxed single float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified.

The first successor is chosen if the first input is less than or equal to the second one; otherwise the second successor is chosen.

Figure 6.61 shows the Graphviz illustration of the `single-float-not-greater-instruction`

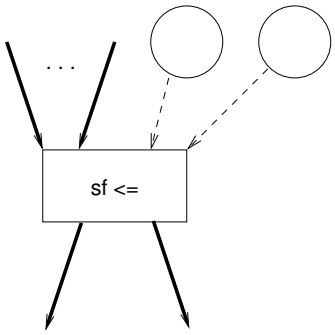

Figure 6.61: `single-float-not-greater-instruction`.

6.3.62 Instruction `single-float-sin-instruction`

Number of inputs	1
Number of outputs	1
Number of predecessors	any
Number of successors	1

The input to this instruction must be of type *unboxed single float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The output is also of type unboxed single float.

The output is the sine of the input.

Figure 6.62 shows the Graphviz illustration of the `single-float-sin-instruction`

6.3.63 Instruction `single-float-cos-instruction`

Number of inputs	1
Number of outputs	1
Number of predecessors	any
Number of successors	1

The input to this instruction must be of type *unboxed single float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The output is also of type unboxed single float.

The output is the cosine of the input.

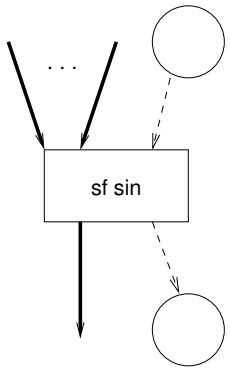

Figure 6.62: `single-float-sin-instruction`.

Figure 6.63 shows the Graphviz illustration of the `single-float-cos-instruction`

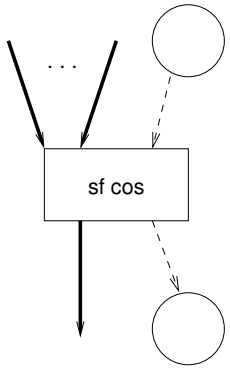

Figure 6.63: `single-float-cos-instruction`.

6.3.64 Instruction `single-float-sqrt-instruction`

Number of inputs	1
Number of outputs	1
Number of predecessors	any
Number of successors	1

The input to this instruction must be of type *unboxed single float*. If the MIR program

is *safe*, then control must reach this instruction only if this restriction is verified. The output is also of type unboxed single float.

The output is the square root of the input.

Figure 6.64 shows the Graphviz illustration of the `single-float-sqrt-instruction`

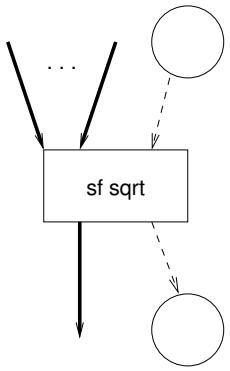

Figure 6.64: `single-float-sqrt-instruction`.

6.3.65 Instruction `double-float-add-instruction`

Number of inputs	2
Number of outputs	1
Number of predecessors	any
Number of successors	1

The inputs to this instruction must be of type *unboxed double float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The output is also of type unboxed double float.

The output is the sum of the two inputs.

Figure 6.65 shows the Graphviz illustration of the `double-float-add-instruction`

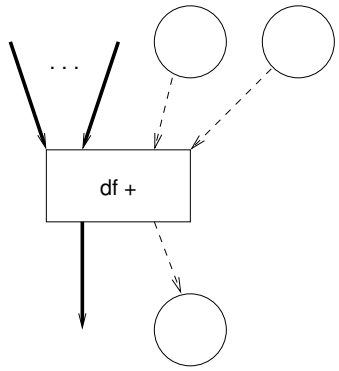

Figure 6.65: `double-float-add-instruction`.

6.3.66 Instruction `double-float-sub-instruction`

Number of inputs	2
Number of outputs	1
Number of predecessors	any
Number of successors	1

The inputs to this instruction must be of type *unboxed double float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The output is also of type unboxed double float.

The output is the difference between the two inputs.

Figure 6.66 shows the Graphviz illustration of the `double-float-sub-instruction`

6.3.67 Instruction `double-float-mul-instruction`

Number of inputs	2
Number of outputs	1
Number of predecessors	any
Number of successors	1

The inputs to this instruction must be of type *unboxed double float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The output is also of type unboxed double float.

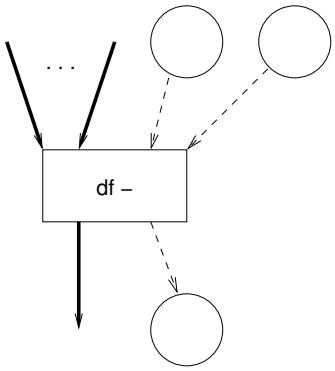

Figure 6.66: `double-float-sub-instruction`.

The output is the product of the two inputs.

Figure 6.67 shows the Graphviz illustration of the `double-float-mul-instruction`

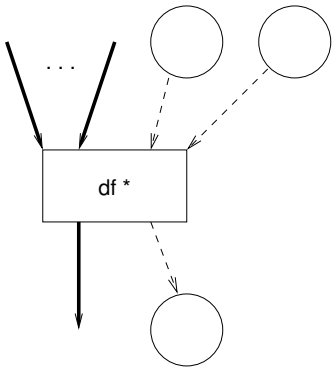

Figure 6.67: `double-float-mul-instruction`.

6.3.68 Instruction `double-float-div-instruction`

Number of inputs	2
Number of outputs	1
Number of predecessors	any
Number of successors	1

The inputs to this instruction must be of type *unboxed double float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The output is also of type unboxed double float.

The output is the quotient of the two inputs.

Figure 6.68 shows the Graphviz illustration of the `double-float-div-instruction`

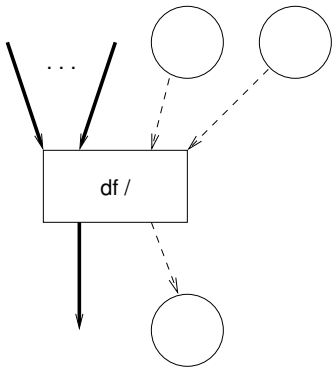

Figure 6.68: `double-float-div-instruction`.

6.3.69 Instruction `double-float-less-instruction`

Number of inputs	2
Number of outputs	0
Number of predecessors	any
Number of successors	2

The inputs to this instruction must be of type *unboxed double float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified.

The first successor is chosen if the first input is strictly less than the second one; otherwise the second successor is chosen.

Figure 6.69 shows the Graphviz illustration of the `double-float-less-instruction`

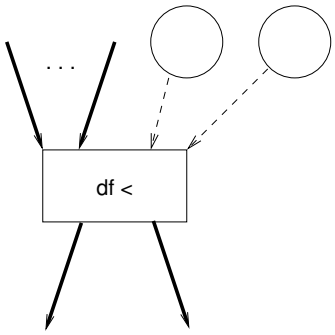

Figure 6.69: `double-float-less-instruction`.

6.3.70 Instruction `double-float-not-greater-instruction`

Number of inputs	2
Number of outputs	0
Number of predecessors	any
Number of successors	2

The inputs to this instruction must be of type *unboxed double float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified.

The first successor is chosen if the first input is less than or equal to the second one; otherwise the second successor is chosen.

Figure 6.70 shows the Graphviz illustration of the `double-float-not-greater-instruction`

6.3.71 Instruction `double-float-sin-instruction`

Number of inputs	1
Number of outputs	1
Number of predecessors	any
Number of successors	1

The input to this instruction must be of type *unboxed double float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The output is also of type unboxed double float.

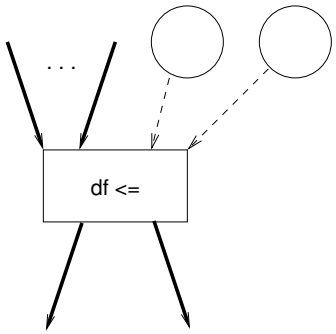

Figure 6.70: `double-float-not-greater-instruction`.

The output is the sine of the input.

Figure 6.71 shows the Graphviz illustration of the `double-float-sin-instruction`

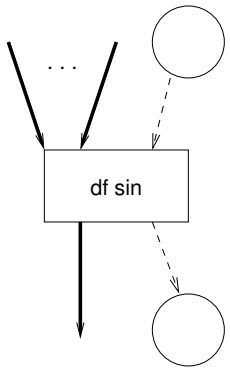

Figure 6.71: `double-float-sin-instruction`.

6.3.72 Instruction `double-float-cos-instruction`

Number of inputs	1
Number of outputs	1
Number of predecessors	any
Number of successors	1

The input to this instruction must be of type *unboxed double float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The output is also of type unboxed double float.

The output is the cosine of the input.

Figure 6.72 shows the Graphviz illustration of the `double-float-cos-instruction`

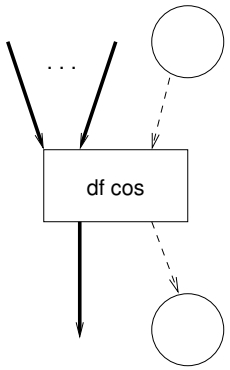

Figure 6.72: `double-float-cos-instruction`.

6.3.73 Instruction `double-float-sqrt-instruction`

Number of inputs	1
Number of outputs	1
Number of predecessors	any
Number of successors	1

The input to this instruction must be of type *unboxed double float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The output is also of type unboxed double float.

The output is the square root of the input.

Figure 6.73 shows the Graphviz illustration of the `double-float-sqrt-instruction`

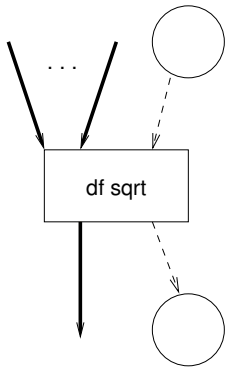

Figure 6.73: `double-float-sqrt-instruction`.

6.3.74 Instruction `long-float-add-instruction`

Number of inputs	2
Number of outputs	1
Number of predecessors	any
Number of successors	1

The inputs to this instruction must be of type *unboxed long float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The output is also of type unboxed long float.

The output is the sum of the two inputs.

Figure 6.74 shows the Graphviz illustration of the `long-float-add-instruction`

6.3.75 Instruction `long-float-sub-instruction`

Number of inputs	2
Number of outputs	1
Number of predecessors	any
Number of successors	1

The inputs to this instruction must be of type *unboxed long float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The output is also of type unboxed long float.

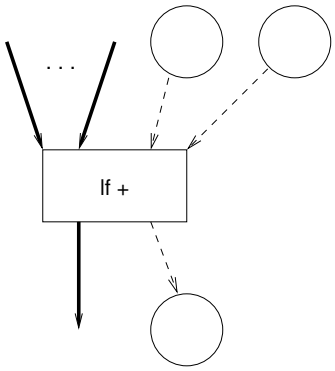

Figure 6.74: `long-float-add-instruction`.

The output is the difference between the two inputs.

Figure 6.75 shows the Graphviz illustration of the `long-float-sub-instruction`

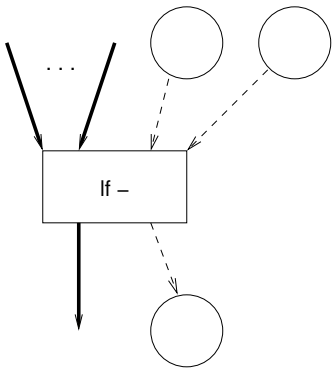

Figure 6.75: `long-float-sub-instruction`.

6.3.76 Instruction `long-float-mul-instruction`

Number of inputs	2
Number of outputs	1
Number of predecessors	any
Number of successors	1

The inputs to this instruction must be of type *unboxed long float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The output is also of type unboxed long float.

The output is the product of the two inputs.

Figure 6.76 shows the Graphviz illustration of the `long-float-mul-instruction`

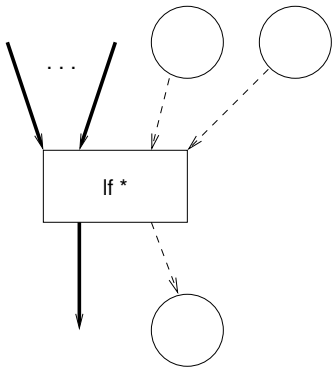

Figure 6.76: `long-float-mul-instruction`.

6.3.77 Instruction `long-float-div-instruction`

Number of inputs	2
Number of outputs	1
Number of predecessors	any
Number of successors	1

The inputs to this instruction must be of type *unboxed long float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The output is also of type unboxed long float.

The output is the quotient of the two inputs.

Figure 6.77 shows the Graphviz illustration of the `long-float-div-instruction`

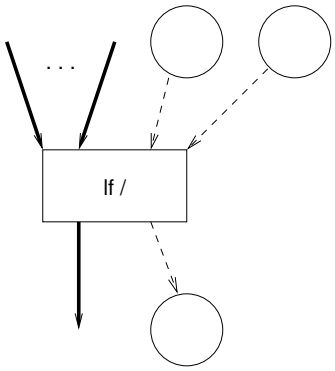

Figure 6.77: `long-float-div-instruction`.

6.3.78 Instruction `long-float-less-instruction`

Number of inputs	2
Number of outputs	0
Number of predecessors	any
Number of successors	2

The inputs to this instruction must be of type *unboxed long float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified.

The first successor is chosen if the first input is strictly less than the second one; otherwise the second successor is chosen.

Figure 6.78 shows the Graphviz illustration of the `long-float-less-instruction`

6.3.79 Instruction `long-float-not-greater-instruction`

Number of inputs	2
Number of outputs	0
Number of predecessors	any
Number of successors	2

The inputs to this instruction must be of type *unboxed long float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified.

The first successor is chosen if the first input is less than or equal to the second one;

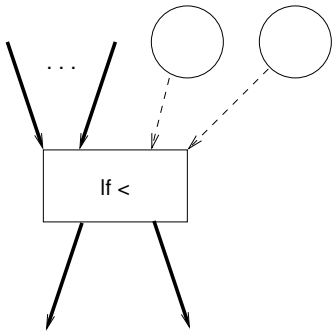

Figure 6.78: `long-float-less-instruction`.

otherwise the second successor is chosen.

Figure 6.79 shows the Graphviz illustration of the `long-float-not-greater-instruction`

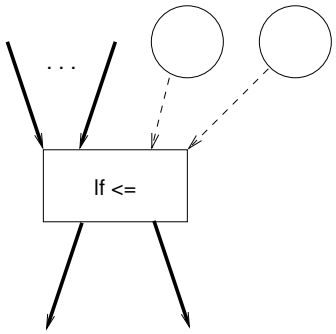

Figure 6.79: `long-float-not-greater-instruction`.

6.3.80 Instruction `long-float-sin-instruction`

Number of inputs	1
Number of outputs	1
Number of predecessors	any
Number of successors	1

The input to this instruction must be of type *unboxed long float*. If the MIR program

is *safe*, then control must reach this instruction only if this restriction is verified. The output is also of type unboxed long float.

The output is the sine of the input.

Figure 6.80 shows the Graphviz illustration of the `long-float-sin-instruction`

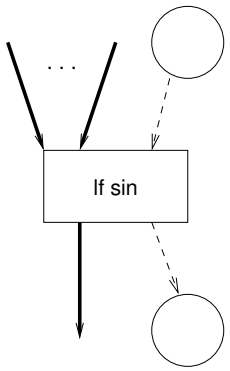

Figure 6.80: `long-float-sin-instruction`.

6.3.81 Instruction `long-float-cos-instruction`

Number of inputs	1
Number of outputs	1
Number of predecessors	any
Number of successors	1

The input to this instruction must be of type *unboxed long float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The output is also of type unboxed long float.

The output is the cosine of the input.

Figure 6.81 shows the Graphviz illustration of the `long-float-cos-instruction`

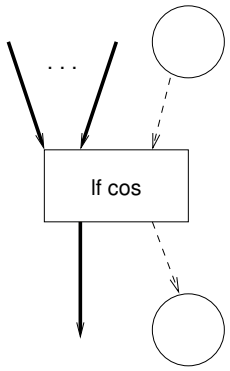

Figure 6.81: `long-float-cos-instruction`.

6.3.82 Instruction `long-float-sqrt-instruction`

Number of inputs	1
Number of outputs	1
Number of predecessors	any
Number of successors	1

The input to this instruction must be of type *unboxed long float*. If the MIR program is *safe*, then control must reach this instruction only if this restriction is verified. The output is also of type unboxed long float.

The output is the square root of the input.

Figure 6.82 shows the Graphviz illustration of the `long-float-sqrt-instruction`

6.3.83 Instruction `create-cell-instruction`

Number of inputs	0
Number of outputs	1
Number of predecessors	any
Number of successors	1

This instruction creates a *cell* to be used as part of the captured static environment of a closure. The exact nature of the cell that is created is not specified. Most implementations would probably just return a `cons` cell since `cons` cells already exist.

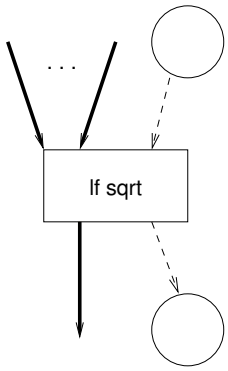

Figure 6.82: `long-float-sqrt-instruction`.

6.3.84 Instruction `fetch-instruction`

Number of inputs	2
Number of outputs	1
Number of predecessors	any
Number of successors	1

This instruction is used when the static environment is represented as a single vector containing an entry for each closed-over variable. Such an entry is either a *cell* (with unspecified representation) if the corresponding variable is modified, or the entry can be the variable itself if it is not modified.

This instruction takes two inputs. The first input is a dynamic lexical location that holds the static environment. The second input is an immediate input containing a non-negative integer and which serves as an index into the static environment. This instruction has a single output, which is a dynamic lexical location.

6.3.85 Instruction `read-cell-instruction`

Number of inputs	1
Number of outputs	1
Number of predecessors	any
Number of successors	1

Since the structure of a cell holding the value of a closed-over variable is unspecified, we use this instruction to obtain the value held in such a cell.

This instruction takes a single input, namely a dynamic lexical location holding the cell to be read from. This instruction has a single output, namely a dynamic lexical location to hold the value of the variable.

6.3.86 Instruction `write-cell-instruction`

Number of inputs	2
Number of outputs	0
Number of predecessors	any
Number of successors	1

Since the structure of a cell holding the value of a closed-over variable is unspecified, we use this instruction to write the value held in such a cell.

This instruction takes two inputs. The first input is a dynamic lexical location holding the cell to be written to. The second input is a constant input or a dynamic lexical input holding the value to write to the cell. This instruction has no outputs.

6.4 Data

6.4.1 Input `constant-input`

This data type corresponds to constants in source code. It can only be used as an *input*.

The slot reader `value` can be used to access the constant.

Figure 6.83 shows the Graphviz illustration of the `constant-input`

Figure 6.83: `constant-input`.

6.4.2 Location `lexical-location`

This data type corresponds to lexical variable in the source code, and to temporary variables introduced by the compiler. It can be used both as an input and as an output.

The slot reader `name` can be used to access the name of the variables. Notice that different lexical locations have different *identity* (i.e, they are not `eq`), but several different lexical locations may have the same name, due to shadowing.

Figure 6.84 shows the Graphviz illustration of the `lexical-location`

Figure 6.84: `lexical-location`.

6.4.3 Location `simple-location`

This data type corresponds is a subtype of `lexical-location`. It is used for lexical locations that are referred to within a single function, so that there is no possible capture. A location of this type can be allocated in a register or on the stack. A temporary variable introduced by the compiler will always turn into a datum of this type.

Figure 6.85 shows the Graphviz illustration of the `simple-location`

Figure 6.85: `simple-location`.

6.4.4 Location `shared-location`

This data type corresponds is a subtype of `lexical-location`. It is used for lexical locations that are referred to from several function, i.e., for *shared* variables. A loca-

tion of this type can *not* be allocated in a register or on the stack. Instead, it has to be allocated in the *static runtime environment*.

Figure 6.86 shows the Graphviz illustration of the `shared-location`

Figure 6.86: `shared-location`.

6.5 Operations on intermediate code

6.5.1 Cloning an instruction

⇒ `clone-instruction` *instruction* [*Generic Function*]

This function creates a copy of *instruction*.

⇒ `clone-instruction` *(instruction* `instruction`*)* [*Method*]

The primary method creates an instance of the class of *instruction*, with the same inputs, outputs and successors (the lists are not copied).

Client code must create `:after` methods to fill in additional slots.

⇒ `insert-instruction-before` *new existing* [*Function*]

Insert the instruction *new* before the instruction *existing*. After this operation *new* will have *existing* as its sole successor, and *existing* will have *new* as its sole predecessor. For every instruction p that was a predecessor of *existing* before this operation was executed, after the operation p will be a predecessor of *new* and *new* will have replaced *existing* as a successor of p.

⇒ `insert-instruction-between` *new existing1 existing2* [*Function*]

Insert the instruction *new* between the instructions *existing1* and *existing2*, where *existing2* is a successor of *existing1*. *existing1* can have any number of successors and *existing2* can have any number of predecessors. *existing1* becomes the sole predecessor of *new* and *existing2* becomes the sole successor of *new*. *new* replaces *existing2* as a successor of *existing1*, and *existing1* as a predecessor of *existing2*.

⇒ `insert-instruction-after` *new existing* [*Function*]

Insert *new* after *existing*. *existing* must have a single successor. The effect is to insert *new* between *existing* and its sole successor.

⇒ `delete-instruction` *instruction* [*Function*]

Delete *instruction*. *instruction* must have a single successor. The sole successor of *instruction* replaces *instruction* as the successor of every predecessor of *instruction*. The predecessors of *instruction* become the predecessors of the sole successor of *instruction*.

Chapter 7

Translating AST to HIR

The translation of an abstract syntax tree (See Chapter 4.) to high-level intermediate representation (See Chapter 6.) is done by an algorithm that is similar to that of CPS-conversion.[1]

As with CPS-conversion, translation makes the control structure explicit. Another similarity is that translation is done from the end of the program to the beginning.

7.1 ASDF system name

The name of the ASDF system that accomplishes the translation of an abstract syntax tree into HIR is called `cleavir-ast-to-hir` and it is located in the file `cleavir-ast-to-hir.asd` in the sub-directory named `AST-to-HIR`.

7.2 Package

The name of the package that contains the names specific to this system is `cleavir-ast-to-hir`. It is defined in the file named `packages.lisp` in the sub-directory named `AST-to-HIR`.

[1] CPS means Continuation Passing Style.

7.3 Compilation context

Translation of a form is accomplished with respect to a *compilation context*. This context contains a *list of lexical variables* to which the values of the translated AST must be assigned. The length of the list corresponds to the number of values required by the context. The context also contains a *list of successors* which represent MIR instructions to which transfer control after evaluation of the current AST.

⇒ `context` [*Class*]

This class is the base class for compilation contexts.

⇒ `:results` [*Initarg*]

⇒ `:successors` [*Initarg*]

⇒ `:speed-value` [*Initarg*]

⇒ `:debug-value` [*Initarg*]

⇒ `:safety-value` [*Initarg*]

⇒ `:space-value` [*Initarg*]

⇒ `:compilation-speed-value` [*Initarg*]

Chapter 8

Optimizations on intermediate representation

8.1 Reducing an instruction graph

As a result of certain local optimizations, it is possible that a well formed instruction graph nevertheless has instructions that are not reachable from the initial instruction. Removing such unreachable instructions is done by *reducing* it.

To reduce an instruction graph, first the set of all reachable instructions is computed. Then, for each reachable instruction i, its predecessors are examined. If i has a predecessor p that is not a member of the set of reachable instructions, then p is removed as described below. Notice that i must have at least one reachable predecessor, so i must have at least two predecessors, one of which is p. We remove p as follows:

- If i is a `phi-instruction`, then determine the position k of p among the predecessors of i. Remove the k:th input from i and the following `phi-instructions` belonging to the same *cluster*. If this operation results in the `phi-instructions` in the cluster having a single input, then replace each `phi-instruction` in the cluster by an `assignment-instruction`. Finally remove the k:th predecessor of i.

- If i is not a `phi-instruction`, simply remove p from the list of predecessors of i.

8.2 Path replication

Certain optimizations and other transformations consist of eliminating redundant tests. These optimizations are of great importance because of the potentially high cost of such tests, especially if the test is in a loop and it is loop invariant. In contrast to other redundant computations, eliminating redundant tests is not just a matter of replacing a computation by a reference to a variable. Instead, the outcome of the test must be remembered by replicating some instructions.

Redundant tests occur frequently, not because the programmer wrote redundant code, but because they are introduced by different Common Lisp operators, especially when the code for these operators is inlined. Consider the two very common Common Lisp functions `car` and `cdr`. These functions are the lowest-level functions on `cons` cells. The code for these functions must first test whether the argument is a `cons` cell. If the user code contains both a call to `car` and a call to `cdr` in the same execution path, then the second such test is redundant since the outcome is the same. Redundant tests occur in other, similar, situations such as tests to determine whether a variable contains a `fixnum` as part of arithmetic operation on that variable.

The main problem we must solve in order to remove these redundant tests is that the execution paths will merge after the first occurrence of the test, so that the outcome of the test is not known at the point of the second occurrence of the test. Consider the following code that might exist explicitly or implicitly as a result of *destructuring*:

```
(let ((x (car w))
      (y (cdr w)))
  ...)
```

Figure 8.1 shows the equivalent HIR code.

In Figure 8.1, `consp` is an instruction that tests whether the input is a `cons` cell and branches accordingly. The instructions `car` and `cdr` correspond to the Common Lisp operations with the same name, except that they assume that the input has been determined to be a `cons` cell. The cloud-shaped computations correspond to tests for `w` being `nil` and to a call to `error` if those tests fail.

As can be seen in Figure 8.1, before the second `consp` instruction, the execution paths merge, so that at that point it is not known whether `w` contains a `cons` cell or something else. The situation is complicated by the fact that there may be an arbitrary number of instructions between the point where the paths merge and the second `consp` instruction. There can even be arbitrary branching and loops among those instructions. The transformation describe in this section replicates all those

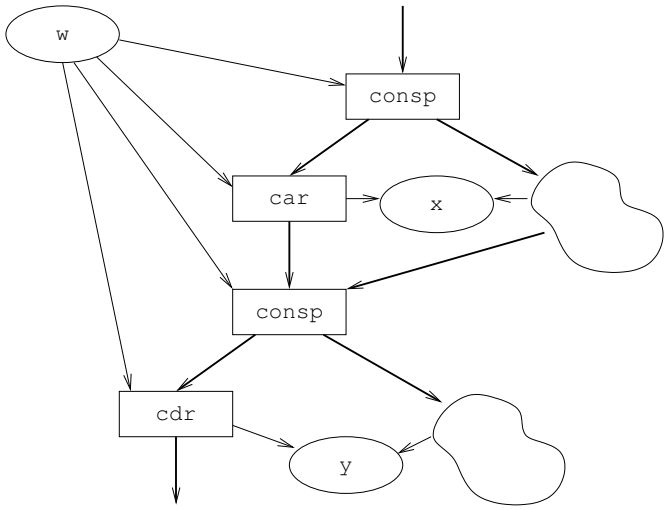

Figure 8.1: Path merging.

instructions so that the second `consp` instruction can be eliminated.

For this transformation to be applicable, there must not be an assignment to the variable being tested between the two tests. Furthermore, the first test must dominate the second test.

The transformation is accomplished by local rewrite rules applying to the second test, or to replicas of that test.

8.3 Static single assignment form

While not an optimization in itself, static single assignment (SSA) form is a prerequisite for many optimization techniques, which is why we describe it here.

In their conference paper from 2013 [BBH+13], Braun et al present an algorithm capable of a direct translation of abstract syntax trees to SSA form. We do not use it here, because we have not been able to couple it with our compilation context which seems to require compilation from the end to the beginning of the program.

Instead, we use the traditional technique based on *iterated dominance frontiers*.

Muchnick [Muc97] discusses SSA form, but the description is very sketchy. He describes using iterated dominance frontiers to find nodes where ϕ functions must be inserted, but he does not discuss how to rename variables. He also does not justify why the start node of the control flow graph is included in the argument to each computation using iterated dominance frontiers. His notation is essentially the same as that of Cytron et al [CFR+91]. For these reasons, we base our description on the paper by Cytron et al, rather than on Muchnick's book.

In fact, there are two papers by Cytron et al that give algorithms for inserting ϕ functions. The first one was published in 1989 in POPL [CFR+89] and the second one in 1991 in TOPLAS [CFR+91].

All research we have seen related to SSA form assumes that functions can not be nested. It is not even clear what it would mean to convert the following code fragment to SSA:

```
(let ((x ...))
  (flet ((f ()
            (incf x)))
    (f)
    (f)))
```

We can see no direct way to convert the body of the local function `f` to SSA form.[1]

Until further research proves otherwise, we must consider SSA to be a per-variable property, so that some variables can not be considered candidates for SSA transformation.

Having said that, we can apply some *tricks* to make it work. In the program previous program fragment, we could alter the signature of the local function `f` as follows:

```
(let ((x1 ...))
  (flet ((f (x)
            (+ x 1)))
    (let ((x2 (f x1)))
      (f x2))))
```

This trick is called *lambda lifting* and it consists of adding extra arguments to a function in order that it can be defined in the null lexical environment.

[1] If the function `f` is *inlined*, it is not hard of course.

Since lambda lifting alters the signature of the function, it can not be used when the function can be called from an external context that assumes the original signature.

Consider the following program fragment:

```
(let ((x 0))
   (find-if (lambda (y) (incf x) (> y 10)) some-sequence)
   ...)
```

This program fragment counts the number of occurrences in `some-sequence` that precede the first occurrence greater than 10. Let's say that following the call to `find-if`, the variable `x` is used in many complex ways so that it is justified to apply SSA conversion to it. Then we can use the following trick:

```
(let ((x1 0))
  (flet ((f (z)
           (find-if (lambda (y) (incf z) (> y 10)) some-sequence)
           z))
    (let ((x2 (f x1)))
      ...)))
```

The trick consists of making a local version of `x`, named `z` and have the closure update that local version instead. Now, of course, `z` can not be SSA converted, but `x` can as the transformed code fragment shows. The reason this trick works is that we know that `find-if` does not hold on to the closure after it returns. We know that it calls the closure a certain number of times, and then it returns without keeping a reference to it.[2]

We can see this trick as a transformation followed by a lambda lifting. The first transformation consists of introducing the local function `f` as follows:

```
(let ((x 0))
  (flet ((f ()
           (find-if (lambda (y) (incf x) (> y 10)) some-sequence)))
    (f)
    ...))
```

[2] FIXME: Find out whether there is a commonly used name for such functions, and if so use it. Otherwise, invent one.

Now, lambda-lifting the function `f` and converting to SSA gives the final result.

However, there are some situations where we can not apply SSA transformation to variables:[3]

Consider the following program fragment:

```
(let ((x ...))
   (f (lambda () (incf x)))
   (g (lambda () (incf x))))
```

In this program fragment, `f` and `g` are arbitrary user-defined functions that may change after the fragment is compiled, so we can not make any assumptions about them. In particular, it is entirely possible that `f` and `g` both capture the closure in global variables like this:

```
(defun f (closure)
  (setf *f* closure))

(defun g (closure)
  (setf *g* closure))
```

In order for this program fragment to respect the semantics of Common Lisp it can be argued that both closures must update a common variable. It follows that both closures must assign to this common variable, and thus that this variable can not be assigned to in a single place as required by SSA.

Some optimizations are possible even if not every variable can be converted into SSA form, so that some variables are assigned multiple times.

In order for a variable V to respect the SSA property, a ϕ function for V is required in control flow node Z whenever there are control flow nodes X and Y containing assignments to V, $X \neq Y$, $X \rightarrow^+ Z$, $Y \rightarrow^+ Z$. The node Z is called a *join point* for V. Since adding a ϕ function in Z introduces an assignment to V, Z must then recursively be considered in order to find other join points requiring additional ϕ functions. The concept of a join point is more generally defined for an arbitrary set of control flow nodes S. The set of join points $J(S)$ is defined as the set of all nodes Z

[3]We should be more specific here: There are some situations where we can not see how it could be possible to apply SSA transformation. Future research into new techniques might come up with some version of SSA that has the desired properties and that will lend itself to these situations.

such that there are two non-null paths starting from two distinct nodes X and Y in S and converge in Z. The *iterated join* $J^+(S)$ is defined as the limit of the increasing sets of nodes $J_1(S) = J(S)$, $J_{i+1}(S) = J(S \cup J_i(S))$.

Consider the control flow graph in Figure 8.2. Empty boxes contain neither assignments nor references to the variable `x`. There are three nodes containing assignments to `x`, namely B, C, and D. The set $J^+(\{B, C, D\}) = \{E\}$.

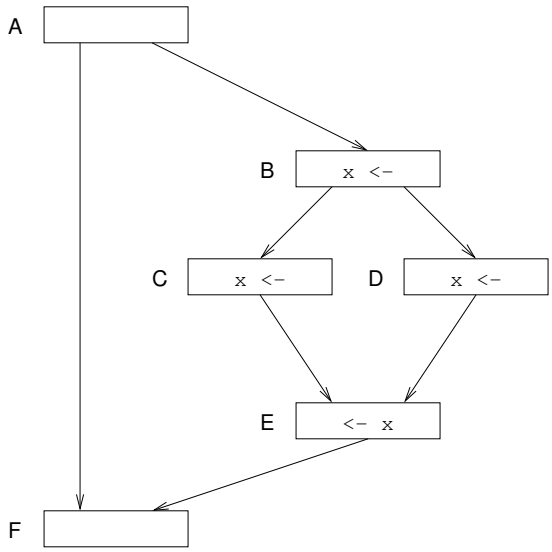

Figure 8.2: Example of join point.

Now, to compute the nodes where a ϕ function is required, Cytron et al use the concept of *iterated dominance frontier*. First, the dominance frontier of a single node x, written $DF(x)$ is the set of nodes y such that x dominates an immediate predecessor of y but x does not strictly dominate y. The dominance frontier of a *set* of nodes S is the natural extension of DF to a set, namely the union of the vales for each element. The iterated dominance frontier is defined in a way similar to the iterated join. $DF^+(S)$ is defined as the limit of the increasing sets of nodes $DF_1(S) = DF(S)$, $DF_{i+1}(S) = DF(S \cup DF_i(S))$.

Consider again the control flow graph in Figure 8.2. The set $DF^+(\{B, C, D\}) = \{E, F\}$. Note that this set includes the node F which is not a join point for the nodes that assign to x. This discrepancy is clearly indicated in the the paper by Cytron et al. If we analyze it a bit more, we see that clearly no ϕ function is required in F. The method using iterated dominance frontiers thus computes more nodes than required.

However, notice that if Figure 8.2 represents a Common Lisp program, then the node F could not contain a reference to `x`, simply because no Common Lisp construct allows the creation of an uninitialized lexical variable. Therefore, in F, the variable `x` must be *dead*. The situation in Figure 8.2 can be quite common in Common Lisp because variables often have very limited scope. It is therefore desirable to avoid considering the node F as needing a ϕ function.

Now consider the control flow graph in Figure 8.3. The only difference from Figure 8.2 is that node E does not contain a use of the variable `x`.

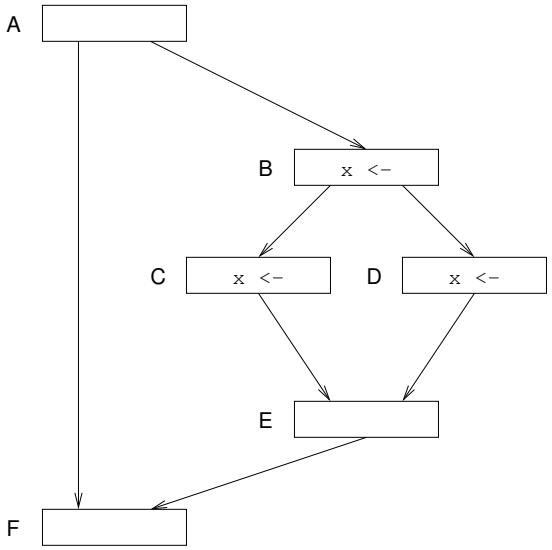

Figure 8.3: Example of join point.

Clearly, the node E is in the set $J^+(\{B, C, D\})$, even though the variable `x` is dead in E. Hence, it is not sufficient to consider an algorithm based directly on join sets rather than on iterated dominance frontiers in order to avoid including nodes where variables are dead. However, checking variable liveness before including a node in the calculation of the iterated dominance frontier takes care of problems generated by both the situation illustrated by Figure 8.2 and by Figure 8.3. A paper by Choi et al [CCF91] describes how to modify the algorithm for placing ϕ functions so that only nodes where the variable is live are affected. In fact the paper by Choi et al describes a modification to the algorithm in the POPL paper by Cytron et all [CFR$^+$89], and the TOPLAS paper by Cytron et all [CFR$^+$91] refers to the paper by Choi et al. The algorithm used in Cleavir is the modified algorithm, except that liveness is determined by a function passed as an argument. Passing `(constantly t)` as an argument will

yield the ordinary SSA form and passing a true liveness test will yield the *pruned* SSA form.

To compute the immediate dominator of every node in the flow graph, we use the algorithm by Langauer and Tarjan [LT79]. They present two versions of their algorithm; one that with complexity $O(m \log n)$ time and one with complexity $O(m \alpha(m, n))$ time, where m is the number of edges in the flow graph, n is the number of nodes, and α is the inverse of Ackermann's function. A more complicated algorithm that runs in linear time was discovered by Harel [Har85].

8.4 Type inference

Type inference is accomplished by formulating it as a *forward data flow* problem. The best result is obtained for variables that respect the *static single assignment* property as described in Section 8.3.

Every *arc* of the flow graph is associated with a type map for each lexical variable that is *live* at that arc. The type map describes all possible types for the variable at that arc. In many cases, the type map can not be precise. If so, then it must contain a *superset* of the possible types. The exact representation of the type map is implementation-dependent.

Initially a fictive arc occurring before the initial `enter-instruction` of the program is considered to have no type maps associated with it, since no variables are live there.

The propagation of type maps depends on the type of each instruction. In particular, the `typeq-instruction` propagates the *and* of the existing type map for a variable and the type specifier of its second input to the first successor arc, and the *difference* to the second successor arc.

When a `typeq-instruction` has an outgoing arc containing the type `nil` for some variable, this means that control can not pass through that arc. We say that such an arc is *dead*. The dead arc becomes the outgoing arc of a `nop-instruction` without a predecessor, and the `typeq-instruction` itself is replaced by a `nop-instruction`.[4] Notice that dead arcs can not simply be removed, because the result would then be an instruction graph that is not *well formed* as described in Chapter 6. After this type of transformation, the instruction graph should be *reduced*. (See Section 8.1.)

[4] FIXME: This explanation is not great. Improve it!

Chapter 9

Backends

9.1 MIR interpreter

Cleavir provides a backend that can run in a Common Lisp system, either intrinsically or extrinsically. This backend is mainly used in order to test that generated MIR code yields the expected result.

The MIR interpreter works as follows:

- The MIR code is translated to a Common Lisp program. This program contains a lambda expression for each nested function. The outermost lambda expression has no parameters.
- The lambda expression is converted to a function using `compile`.
- The resulting function is executed by passing it to `funcall`.

Each nested function has the following shape:

- The outermost form is a lambda expression with a lambda list of a particular form. See below for details about the lambda list.
- The body of the lambda expression is a `let` special form. The variables of the `let` form are all the lexical variables used by the function, but not used by an enclosing function. The `let` variables are all initialized to `nil`.

- The body of the `let` is a `block` special form. The name of the block is `nil`. This block is used in the translation of the `return` instruction.
- The body of the block is a `tagbody` special form. Each instruction generates a statement of the tagbody. A statement may be preceded by a `tag` if there is a `go` to that statement.

The lambda list of the function has no initialization forms. Here are the details of what it contains.

- It has zero or more required parameters. Each required parameter is a symbol naming a variable.
- It may have `&optional` parameters. Each such parameter is a list of two symbols. The first symbol names the main parameter. The second symbol names a *supplied-p* parameter. The latter is included even when the original lambda list did not contain it. The body of the function tests the value of the *supplied-p* parameter in order to determine whether to execute some initialization code that assigns a default value to the main parameter.
- It may have a `&rest` parameter. This parameter is just a symbol.
- It may have `&key` parameters. Each such parameter is a list of three symbols. The first symbol names the key to be searched for in the argument list to find the argument. The second symbol names the main parameter. The third symbol names a *supplied-p* parameter. As with `&optional` parameters, the latter is included even when the original lambda list did not contain it. Again, the body of the function tests the value of the *supplied-p* parameter in order to determine whether to execute some initialization code that assigns a default value to the main parameter.
- It may contain the lambda-list keyword `&allow-other-keys`.

Chapter 10

Metering

The system named `cleavir-meter` defines a package with the same name, and functionality for *metering*.

Metering is an idea inspired by the Multics operating system, and consists of accumulating performance information whenever doing so does not noticeably degrade performance of the subsystem being metered. The performance information consists at the very least of counting the number of invocations of a function and the processor time the invocations took to execute.

⇒ `meter` [*Class*]

This class is the base class for all meters.

⇒ `basic-meter` [*Class*]

This class is a subclass of the class `meter`. Meters of this type collect information about processor time and number of invocations in a way that make it possible to determine minimum and maximum processor time for the invocations as well as the standard deviation for the processor time.

⇒ `size-meter` [*Class*]

This class is a subclass of the class `basic-meter`. In addition to counting the number of invocations and measuring processor time, this class allows client code to associate a *size* with each invocation.

⇒ `reset` *meter* [*Generic Function*]

This generic function uses the `progn` method combination. It resets all information

to its initial value.

⇒ `report` *meter* `&optional` *stream* [*Function*]

This function generates a report on `stream`. The `stream` argument defaults to the value of the special variable `*standard-output*`.

⇒ `with-meter` *(meter-variable meter-form)* `&body` *body* [*Macro*]

This macro evaluates *body* in a context where *meter-variable* is bound to the value of the form *meter-form*. Some simple processing is done automatically, such as noting the processor time before and after the evaluation of the forms in *body*. More complex processing can be done by some form in *body* by referring to *meter-variable*.

This macro accomplishes its task by wrapping the forms in *body* in a *thunk* and calling the generic function `invoke-with-meter` with *meter-variable* and the thunk as arguments.

⇒ `invoke-with-meter` *meter function* [*Generic Function*]

This generic function is called from the expansion of the macro `with-meter` with the meter variable and *thunk* containing the body of the macro call.

Bibliography

[BBH+13] Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. Simple and efficient construction of static single assignment form. In *Proceedings of the 22Nd International Conference on Compiler Construction*, CC'13, pages 102–122, Berlin, Heidelberg, 2013. Springer-Verlag.

[CCF91] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 55–66, New York, NY, USA, 1991. ACM.

[CFR+89] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 25–35, New York, NY, USA, 1989. ACM.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[Har85] D Harel. A linear algorithm for finding dominators in flow graphs and related problems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, pages 185–194, New York, NY, USA, 1985. ACM.

[LT79] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, January 1979.

[Muc97] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

Index