

PLT MrSpidey: Static Debugger Manual

Cormac Flanagan
cormac@cs.rice.edu

Edited by: Matthias Felleisen
matthias@cs.rice.edu

Version 102
June 2000

Copyright notice

Copyright ©1995-97 Cormac Flanagan

Permission to make digital/hard copies and/or distribute this documentation for any purpose is hereby granted without fee, provided that the above copyright notice, author, and this permission notice appear in all copies of this documentation.

Zodiac: Copyright ©1995-2000 Shriram Krishnamurthi. All rights reserved.

DrScheme: Copyright ©1996-2000 PLT, Rice University. All rights reserved.

Contents

1	Introduction to MrSpidey	1
1.1	Thanks	1
2	Using MrSpidey	2
2.1	The Program Window	2
2.1.1	Unsafe Operations	2
2.1.2	Popup Menus	2
2.1.3	Type Information	4
2.1.4	The Value Flow Browser	4
2.2	The Summary Window	5
3	Preferences	7
3.1	MrSpidey Analysis Preferences Window	7
3.2	MrSpidey Type Display Preferences Window	9
4	Analysis of Large Programs	11
4.1	Inter-File Arrows	11
5	The Type Language	14
5.1	Accurate Numeric Operations	15
6	Extensions to DrScheme	16
6.1	Type Assertions	16
6.2	Polymorphic Annotations	16
6.3	Declaring New Primitives	16
6.4	Declaring Constructors	16
6.5	Declaring New Types	16

7 Customization	17
8 Restrictions on Source Programs	18

1. Introduction to MrSpidey

MrSpidey is an interactive, static debugger for Scheme designed to help programmers understand and debug complex programs. It automatically infers information about the run-time behavior of programs, and uses this information to identify potential “danger-points” in those programs. Specifically, MrSpidey:

- infers a *type*, or value set invariant, describing the set of possible values for each program expression;
- uses this information to identify *unsafe* program operations that may cause run-time errors; and
- provides a supporting graphical explanation for these invariants.

MrSpidey supports almost all of the constructs and procedures found DrScheme’s R4RS+ language level. This language levels extends R4RS Scheme with structures, a module system, an object system, and a GUI toolbox. For further information on the technology underlying MrSpidey, see [1, 3, 4, 5, 2].

1.1 Thanks

Many thanks to both Matthew Flatt and Robby Findler for MrEd, and to Shriram Krishnamurthi for Zodiac, his source-correlating macro-expander. MrSpidey crucially depends on both of these packages. Thanks also to Stephanie Weirich for work on the first implementation of MrSpidey, and to Matthias Felleisen, Corky Cartwright, Jeremy Buhler, and the Rice University Spring ’96 COMP311 programming languages class for their feedback and help.

The typesetting sources for this manual are taken from *Reference Manual for wxWindows 1.60: a portable C++ GUI toolkit*.

2. Using MrSpidey

MrSpidey is an integrated portion of DrScheme. To analyze the current program in DrScheme, click on DrScheme's **Analyze** button. MrSpidey analyzes the program and displays the analysis results in a new frame. This frame contains two sub-windows: a program window and a summary window. The display of these windows is controlled via the **Show** menu.

2.1 The Program Window

The program window contains an annotated version of original program. The additional annotations present information about the results of the analysis, as described below.

2.1.1 Unsafe Operations

An operation is (potentially) *unsafe* if it may be applied to inappropriate arguments during an evaluation, thus raising an error. Since unsafe operations are natural starting points for static debugging, MrSpidey highlights them via font and color changes as follows:

- Any primitive operation that may be applied to inappropriate arguments, thus raising a run-time error, is highlighted in red (or underlined on monochrome screens).¹ Conversely, primitive operations that never raise errors are shown in green.
- Any function that may be applied to an incorrect number of arguments is highlighted by displaying the `lambda` keyword in red (or underlined).
- Any application expression where the function position may return a non-function is highlighted by displaying the enclosing parentheses in red (or underlined).

Figure 2.1 contains examples of these three different kinds of unsafe operations. The **tab** key moves the focus forward to the next unsafe operation, and the **shift-tab** key moves the focus backward to the previous unsafe operation. These keystrokes make it easy to inspect the unsafe operations in a program.

2.1.2 Popup Menus

MrSpidey also computes significant additional information for each analyzed expression. This information is available on a demand-driven basis via pop-up menus. MrSpidey associates a pop-up menu with all program variables, which are marked in bold, and also with the opening parenthesis of each expression, which is also marked in bold: see figure 2.2. Clicking on one of these bold tokens displays the associated menu, which then provides access to additional type and value flow information, as described below.

¹Certain unsafe operations, such as a **vector-ref** operation whose index argument may be out-of-range, are not detected. A list of such operations is contained in chapter 8.

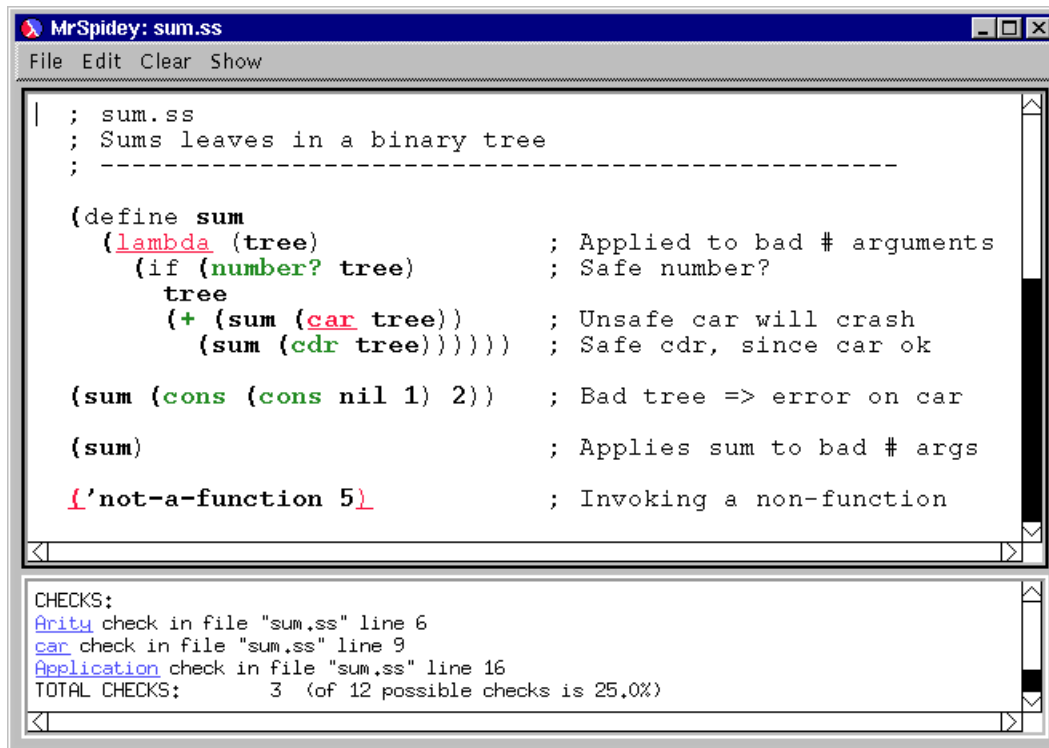


Figure 2.1: Identifying unsafe operations

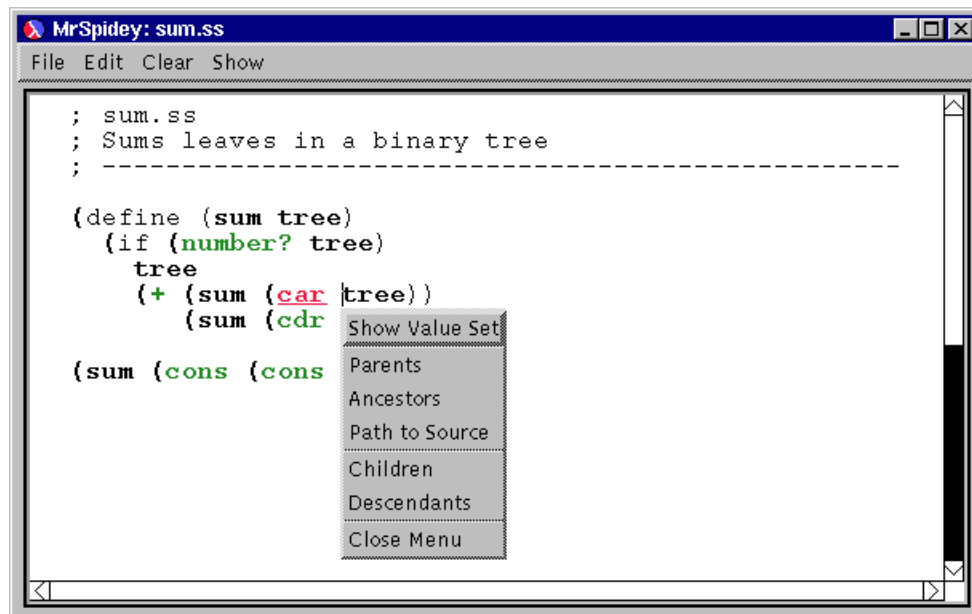


Figure 2.2: The pop-up menu

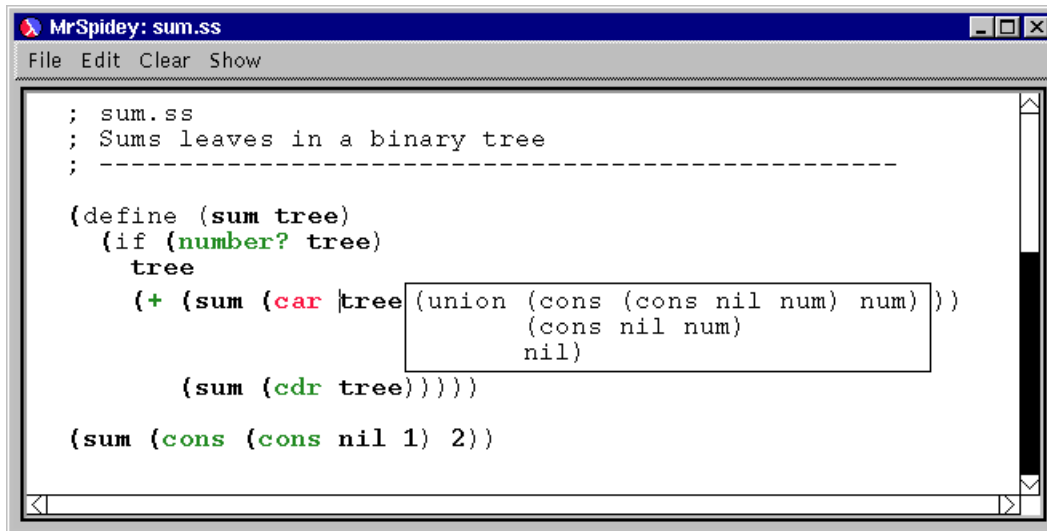


Figure 2.3: Displaying type information

2.1.3 Type Information

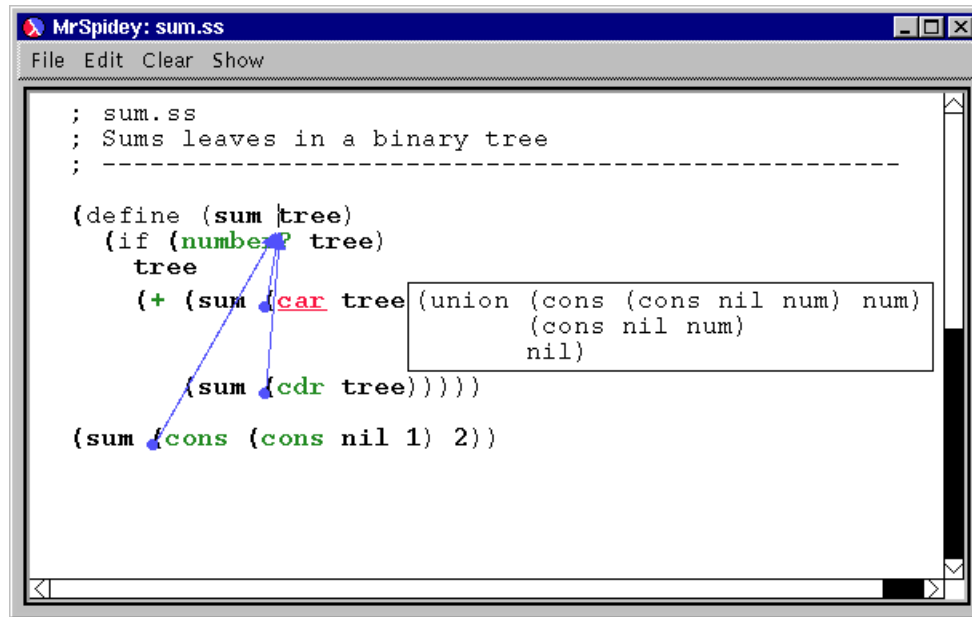
MrSpidey provides an inferred type for each program expression. To view the type of an expression, select the **Show Value Set** option of the expression's menu. Mrspidey then computes the expression's type and displays it in a box inserted to the right of the expression, as illustrated in figure 2.3. See Section 5 for a complete description of the type language. The type box is deleted by selecting the **Close Value Set** option from the popup menu. Alternatively, selecting **Clear Types** deletes all type boxes in the buffer.

2.1.4 The Value Flow Browser

MrSpidey can also explain the derivation of each value set invariant, or type. This explanation describes how the flow of values through the program yields a particular value set invariant. The collection of all potential paths along which value may flow through the program forms the program's *value flow graph*. MrSpidey describes each edge in the data-flow graph as an arrow overlaid on the program text that connects the relevant points of the program. Because a large number of arrows would clutter the program text, these arrows are presented in a demand-driven fashion. Each expression's popup menu provides facilities for inspecting relevant portions of the value flow graph.

- **Parents**: exposes arrows indicating the immediate parents of the current expression in the value flow graph. For example, figure 2.4 shows the incoming edges for the parameter `tree` in the program `sum`.
- **Ancestors**: exposes all ancestors of the current expression in the value flow graph.
- **Path to Source**: finds the shortest path in the value flow graph from a constructor expression to the current expression.
- **Children**: exposes the immediate children of the current expression in the value flow graph.
- **Descendants**: exposes all descendants of the current expression in the value flow graph.

All of the above options can be customized to show only the flow of certain values by selecting the appropriate set of values from the **Filter** menu. This option is particularly useful in conjunction with the **Path to Source** facility for inspecting the flow of an unexpected value through the program. For example, in the

Figure 2.4: Parents of `tree`

program `sum`, the unexpected value for `tree` is `nil`. Setting the filter to this value, and then selecting **Path to Source** for `tree` results in the explanation described in figure 2.5.

Clicking on the head or tail of an arrow with the left mouse button moves the current focus to the term at the other end of the arrow, which can be useful for following the flow of values through large programs. Clicking on the head or tail of an arrow with the right mouse button deletes the arrow. Alternatively, selecting **Clear|Arrows** deletes all arrows in the buffer.

2.2 The Summary Window

The summary window lists all unsafe operations in the program, together with hyper-links to those operation. It counts the number of unsafe operations, and expresses that number as a percentage of the total number of operations in the program. This window also contains warning about unbound variables and failed type assertions. A typical summary window is show in figure 2.1.

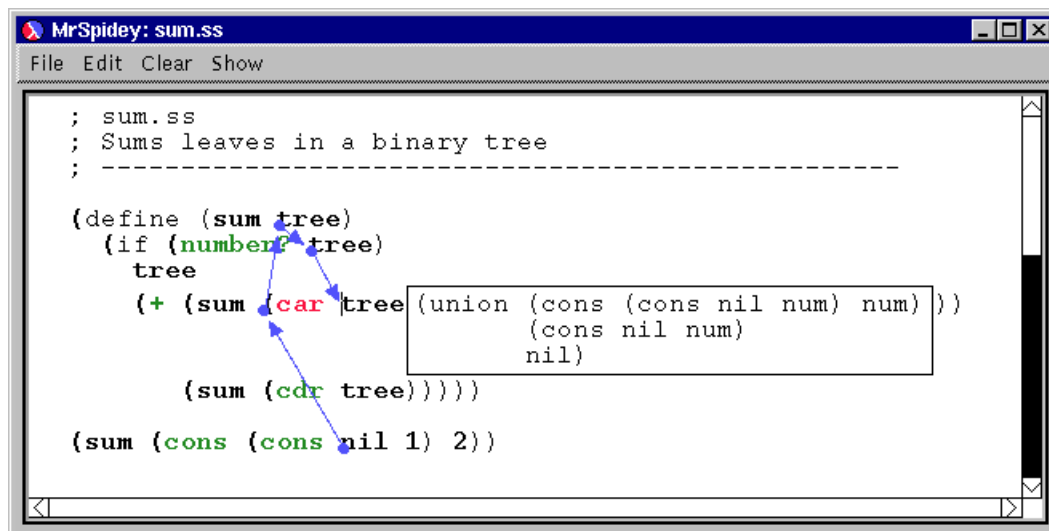


Figure 2.5: Flow of nil

3. Preferences

The `Edit|Preferences ...` menu item menu allows users to configure a variety of DrScheme options. Two of the preferences windows, `MrSpidey Analysis` and `MrSpidey Type Display`, control aspects of MrSpidey's behavior. (These windows are only available after MrSpidey is loaded.)

3.1 MrSpidey Analysis Preferences Window

The `MrSpidey Analysis` preferences window configures MrSpidey's analysis of programs. An example of this window is shown in figure 3.1, and the controls are described below.

- **Accurate constant types:** When this button is off (default), then character, symbolic and numeric constants are given the types `char`, `sym` or `num` respectively. If the button is on, then these constants are typed accurately, *i.e.* the number 4 is assigned the type 4, etc.
- **Constant merge size:** Large quoted values in Scheme can yield large types that significantly increase the analysis time. To overcome this problem, MrSpidey generates *approximate* types for such constants. The **Constant merge size** slider controls how large constants can get before MrSpidey starts to approximate their type.
- **If splitting:** If this button is on (default), MrSpidey analyzes conditional expressions such as `(if (number? x) ...)` in a special manner. Specifically, MrSpidey propagates only numeric values for `x` in the then-part, and non-numbers in the else-part. At the moment, MrSpidey performs if-splitting for conditions that test simple top-level value constructors, e.g., `number?`, `symbol?`, `pair?`, etc.
- **Flow sensitivity:** If this button is on (default), then after an expression such as `(car x)`, MrSpidey knows that the value of `x` must be a pair.
- **Accurate analysis of numeric operations:** When this button is off (default), then numeric operations such as `+` simply return the type `num`. If the button is on, then the numeric operations are typed more accurately, as described in subsection 5.1.
- **Polymorphism:** This radio box controls how polymorphic expressions (see section 6.2) are analyzed.
 - **No polymorphism:** polymorphic annotation is ignored;
 - **Simplify constraints:** The constraint system for the polymorphic expression is simplified (see below); and
 - **Reanalyze:** The polymorphic expression is re-analyzed for each polymorphic reference.
- **Polymorphism simplification algorithms:** This radio box controls which constraint simplification algorithm is used to simplify the constraint system of polymorphic expression, provided the **Polymorphism** control is set to **Simplify constraints**. These constraint simplification algorithms are described in [4].
- **Save .za files in:** MrSpidey generates and saves constraint (`.za`) files during the analysis of multi-file programs. This radio box controls where these constraint files are stored, either in the same directory as the corresponding source file (default), or in a temporary directory.

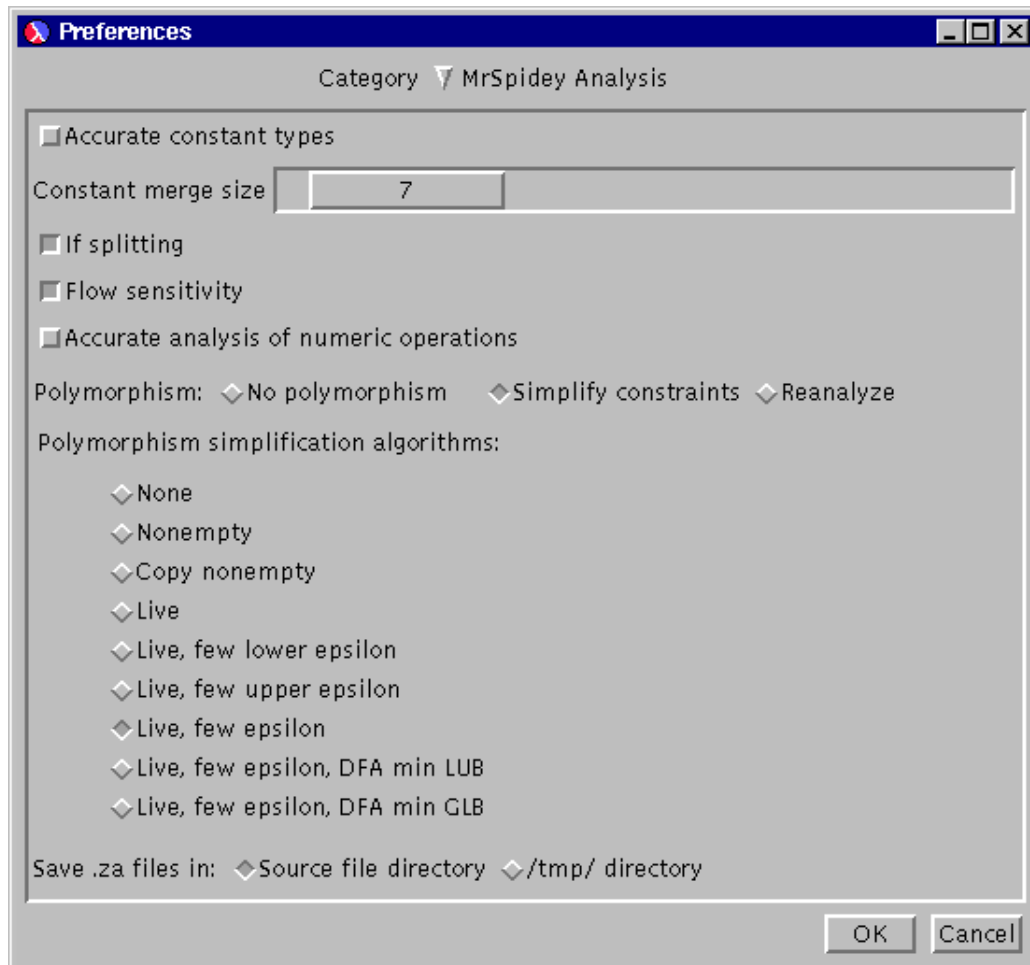


Figure 3.1: MrSpidey Analysis preferences window

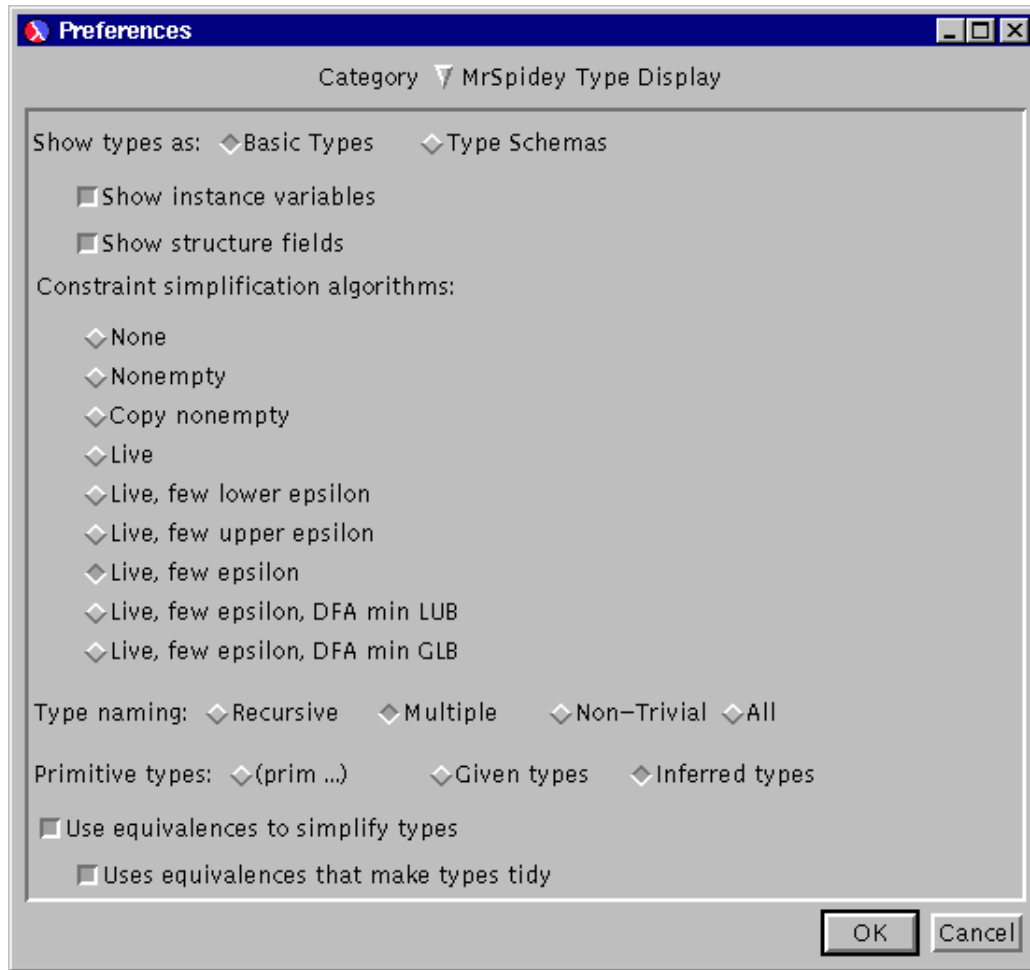


Figure 3.2: MrSpidey Type Display preferences window

3.2 MrSpidey Type Display Preferences Window

The MrSpidey Type Display preference window controls how MrSpidey computes and displays type information. An example of this window is shown in figure 3.2, and the controls are described below.

- **Show types as:** Types can be displayed either as basic types (default), which just show the range of functions, or as type schemas, which show how the domain and range of a function are related (*e.g.* $(X1 \rightarrow X1)$). Basic types contain less information, but are more compact, which is an important benefit when working with large programs. With basic types, users can also choose whether or not to show instance variables and structure fields. For large programs, it is often best not to show these components, in order to produce reasonably compact types.
- **Constraint simplification algorithms:** This radio box controls which constraint simplification algorithm is used to simplify the constraint system of an expression, before converting that constraint system to a type. These constraint simplification algorithms are described in [4].
- **Type naming:** This parameter controls what types are named in a `rec` type expression, and can be:
 - **Recursive:** Names just enough types to express recursive types.
 - **Multiple:** (default) Names every type that is referred to more than once
 - **Non-Trivial:** Names every type except trivial types such as `num`, `sym`, etc.

- **All:** Names all types.
- **Primitive types:** This parameter can be:
 - **(prim ...):** Displays primitive types as `(prim car)` etc.
 - **Given types:** Displays the given types of primitives, *e.g.* `((cons a b) -> a)`.
 - **Inferred types:** (default) Displays the inferred domain and range of primitive functions, *e.g.* `((cons num 4) -> num)`.
- **Use equivalences to simplify types:** If this control is on (default), then a number of rewriting rules are used to simplify types before they are displayed.
- **Use equivalences that make types tidy:** If this control is on (default), then some of the type rewriting rules will merge types into disjoint unions, thus losing a certain amount of type information in order to produce a more compact type. Note: A disjoint union is a **union** type in which all variants have mutually distinct constructors.

4. Analysis of Large Programs

Large programs are typically split into multiple source files, where each source file contains a **unit** (or **unit/sig**) expression. The main file for the program then refers to each source file via the **require-unit** (or **require-unit/sig**) form, and links these multiple units together into a single compound-unit that is then invoked.

To provide a quick turn-around time when statically debugging such programs, MrSpidey uses a *componential* analysis to avoid re-analyzing source files where possible. When each required unit file is first analyzed, MrSpidey:

1. derives a constraint system that describes the data-flow behavior of the unit;
2. simplifies the constraint system while preserving the externally-visible information about the unit's data-flow behavior; and
3. saves the simplified constraint system in a *constraint file* named *file.za* (where *file.ss* is the name of the source file).

Although the use of constraint files does not reduce the time required for the first analysis, on subsequent analyses MrSpidey can use the saved constraint files to avoid the re-analysis of required unit files that have not been modified. This approach substantially reduces re-analysis times.

Once the analysis is completed, MrSpidey displays the program's main file with the usual static debugging mark-ups. To view other source files, select the **Actions|Open ...** option from the MrSpidey window. This option displays a dialog box containing all the program's source files. Select the file of interest. A typical dialog box is contained in figure 4.1. Alternatively:

- The **Actions|Open All** option opens a MrSpidey windows for each source file; and
- The **Actions|Load All** option loads all source files into memory but does not immediately display them.

The **Actions|Close All** option closes all the MrSpidey windows.

4.1 Inter-File Arrows

In multi-file programs, the source (or destination) of an arrow may sometimes refer to a program point in a separate file. In this case MrSpidey draws an arrow originating (or terminating) in the left margin of the program: see figure 4.2. If a *margin arrow* is painted red, then it refers to an expression in a file that has not yet been loaded, and clicking on the arrow provides the option to load the file. A blue margin arrow refers to an expression in a file that has been loaded. Clicking on a blue arrow provides the option to zoom to and highlight the term at the other end of the arrow, as shown in figure 4.3. These facilities are useful for following the flow of values through multi-file programs.

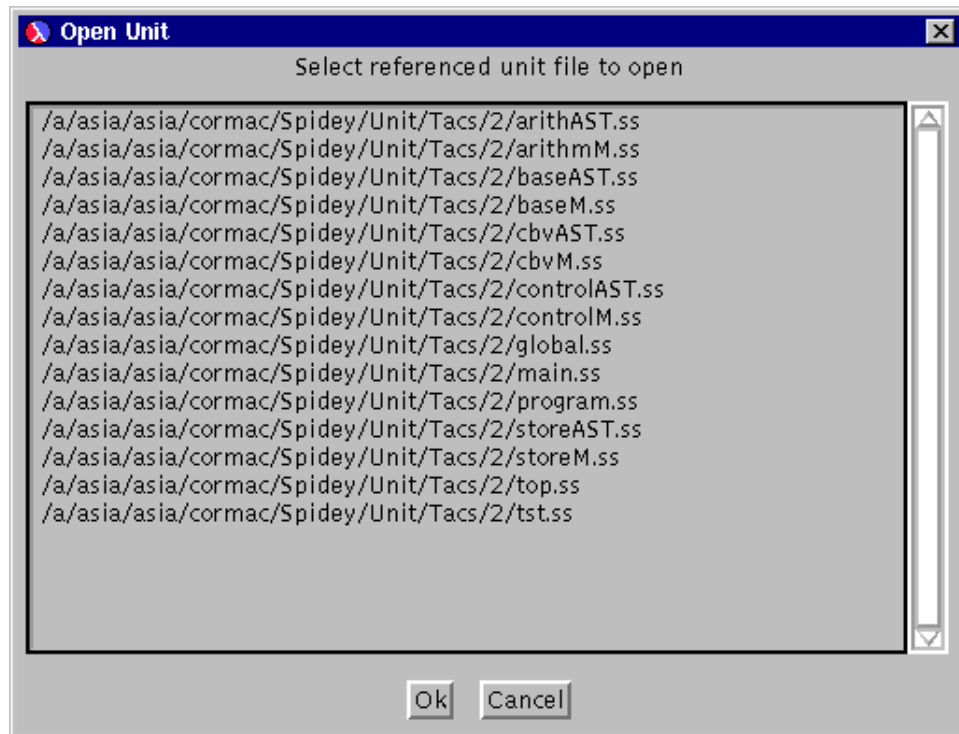


Figure 4.1: The Actions|Open ... dialog box

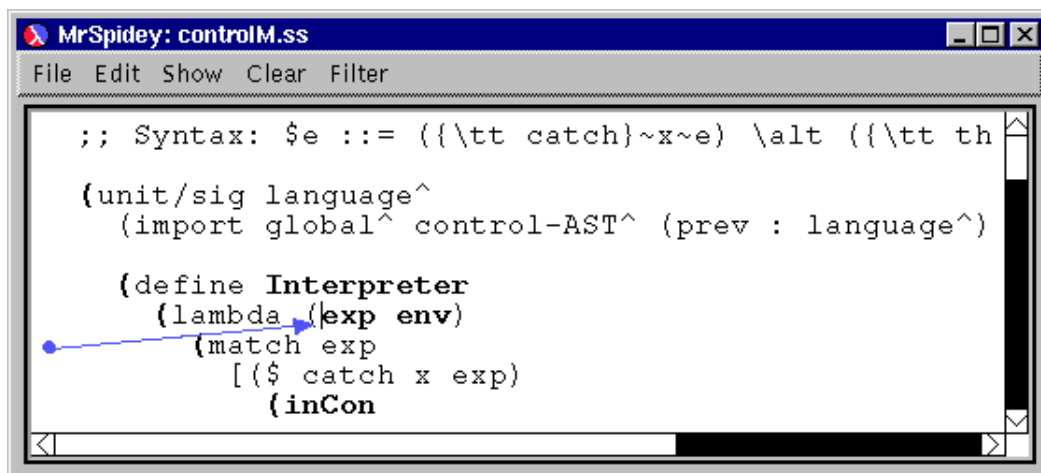


Figure 4.2: Source in another loaded file

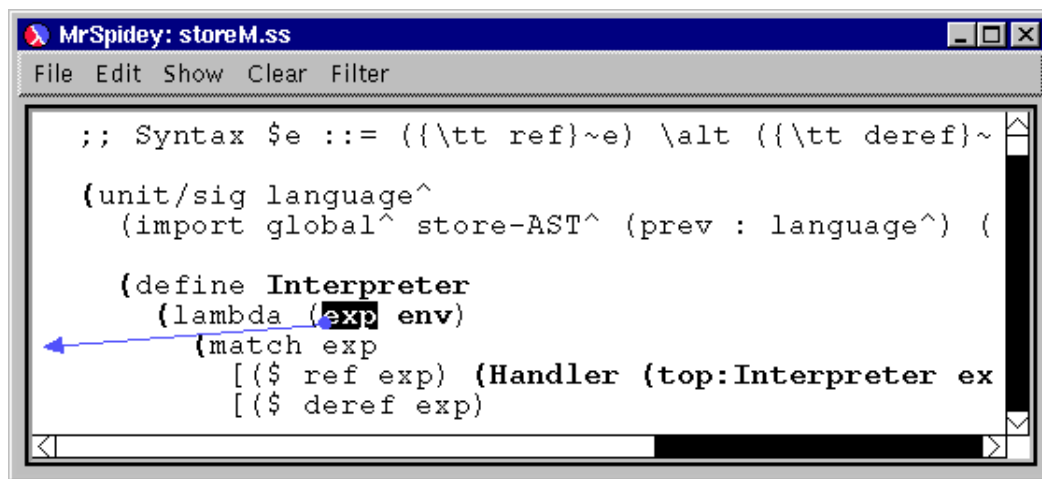


Figure 4.3: The highlighted source in the other file

0 0 1 0 0 1

```
set-variable
```

zeroary-constructor

(*constructor time ... time*)

```
(union time ... time)
```

```
(rec ([set-variable time] ...) time)
```

```
(class [img time] ...)
```

```
(object [ivar time] ...)
```

function-time

time-abbreviation

identifier

```
nil num sym str char void true false bool eof
```

```
zeroarg_constructor
```

```
vec box promise mvalues iport oport      Unary constructors
```

Unary constructors

Binary constructors

```
user-defined-constructor
```

identifier

identifier

$$(time \dots time \rightarrow time)$$

t_1	t_1	t_1
$(t_{\text{une}} \dots t_{\text{une}} \rightarrow t_{\text{une}})$		Best argument

Best argument

Return value list

Post argument and

```

return value list

```

same as $(time \rightarrow* time)$

	Abbreviates
(MU <i>set-variable type</i>)	(rec ([<i>set-variable type</i>]) <i>set-variable</i>)
noarg	nil
(arg <i>type type</i>)	(cons <i>type type</i>)
(list <i>type ... type</i>)	(cons <i>type</i> (cons ... (cons <i>type</i> nil)))
(listof <i>type</i>)	(MU 1 (union (cons <i>type</i> 1) nil))
null	nil
bool	(union true false)
atom	(union nil num sym str char bool)
sexp	(MU x (union atom (cons x x) (vec x)))

The behavior of primitive operations is defined using *multiple-arity schemas*. For each reference to a primitive operation, MrSpidey retrieves the corresponding multiple-arity schema and selects the schema appropriate for the number of arguments given to the primitive (or the last schema if the primitive is used in a higher-order manner). It then instantiates the schema by replacing the quantified set variables by set variables, and converts the resulting basic type into a constraint system. Multiple-arity schemas are defined as follows:

multiple-arity-schema is one of:
schema
 (case-> *schema ...*)

schema is one of:
type
 (forall (*set-variable ...*) *type*)

5.1 Accurate Numeric Operations

When the **Accurate numeric operations** control in the **MrSpidey Analysis** preferences window is turned on, MrSpidey performs a more accurate analysis of numeric operations, as follows.

The type language is extended with the unary constructors **apply+**, **apply-**, **apply*** and **apply/**. The return type of the numeric operations **+**, **-**, *****, and **/** record information about the numeric operation and its argument value sets. For example, the type returned by the operation **+** is (**apply+ arglist**), where *arglist* is the argument list to **+**. The resulting types are simplified before being presented to the programmer. For example, type (**apply+ (list x1 ... xn)**) is transformed into (**+ x1 ... xn**), etc.

In addition, the binary constructors **=**, **not=**, **<**, **<=**, **>** and **>=** are added to the type language. The meaning of the type (**< X Y**) is the set of numbers *x* in *X* such that there exists some *y* in *Y* with *x* < *y*. MrSpidey generates these types for **if**-expressions where the predicate is one of **zero?**, **=**, **<**, **<=**, **>** or **>=**.

6. Extensions to DrScheme

6.1 Type Assertions

The form `(: exp type)` is an assertion that the values produced by *exp* must be contained in *type*. If MrSpidey is unable to prove that the type assertion is satisfied, then a warning is reported in the summary window. A `:`-expression evaluates to void.

6.2 Polymorphic Annotations

The form `(polymorphic exp)` causes the expression *exp* to be analyzed in a polymorphic manner. That is, if the result of `(polymorphic exp)` is immediately bound to an identifier (*e.g.* by **let** or **define**), then all references to that identifier that occur below that binding will be polymorphic. The annotation has no runtime effect.

6.3 Declaring New Primitives

The form `(type: multiple-arity-schema)` declares new primitive of type *multiple-arity-schema*. Some example definitions are:

```
(define my-car (type: (forall (a) ((cons a _) -> a))))
```

```
(define my-map (type: (forall (a r) ((a -> r) (listof a) -> (listof r)))))
```

The expression `(type: ...)` evaluates to void.

6.4 Declaring Constructors

The form `(define-constructor name modes ...)` adds a new type constructor to the type language. The arguments *mode* ... are all booleans, each specifying whether the corresponding field in the constructor is mutable.

6.5 Declaring New Types

The form `(define-type name type)` adds a new type *name* that can later be used in type expressions.

7. Customization

Definitions of the following window manager resource variables (typically defined in the `.Xresources` in the users home directory) allow the user to configure how MrSpidey displays the program and associated static debugging information.

The root for all window manager variables is either “`mrspidey`” or “`mrspidey-bw`”, depending on whether or not the system is using a monochrome monitor. Each `delta` variable described below should be bound to a string containing a series of Scheme lists. Each list should be *(method args ...)*, where *method* is a `wxWindow wx:StyleDelta` method, and *args ...* is an appropriate argument list for that method. A typical definition is:

```
mrspidey.check-delta: (set-delta wx:const-change-underline 1)
```

The list of `delta` variables used by MrSpidey is:

- `base-delta`: The base style for program text and types.
- `normal-delta`: For displaying normal (unannotated) text.
- `type-link-delta`: For text with a hyper-link to the popup menu for showing the type, etc.
- `type-delta`: For showing the type in a box.
- `check-delta`: For showing checks (default red).
- `uncheck-delta`: For showing unchecked primitives (default forest green).
- `check-link-delta`: For hyper-links from the summary window to the corresponding definition.
- `arrow-color`: The color for showing (most) arrows (default blue)

8. Restrictions on Source Programs

The following DrScheme facilities are not handled.

- Primitives for dynamic loading and evaluation: `load`, `load/cd`, `load-relative`, `load/use-compiled`, `current-load`, `require-library-use-compiled`, `compile`, `eval`, `current-eval`, `expand-defmacro` and `expand-defmacro-once`.
- Primitives that dynamically manipulate the top-level environment: `undefine`, `global-defined-value`, `invoke-open-unit` and `invoke-open-unit/sig`.
- MzScheme's interfaces, exception system and exception hierarchy.
- Primitives that access structures without the appropriate selectors: `struct-ref` and `struct->vector`.
- MrSpidey doesn't know about DrScheme language levels.
- Primitive names should not be assigned or defined.

Certain potential errors are not detected:

- Returning an inappropriate number of multiple values to a single value context.
- Index out of bounds on vector, string and list operations.
- `compound-unit` errors. Since `unit`'s are normally used in a first-order manner, these errors are typically easy to detect using the evaluator.
- The arguments to `primitive-name` and `primitive-result-arity` are only checked to be procedures, not primitive procedures.
- Errors in `read` due to ill-formed s-expression on the input port.

The analysis of certain kinds of code is not sound:

- Values passed to exception handlers.
- Tracking values through parameters.
- Tracking values through will executors.
- MrSpidey doesn't translate `unit/sig` into `unit`, as MzScheme does.

In other words, an analysis of the above constructs may produce false information.

Bibliography

- [1] Cormac Flanagan. *Effective Static Debugging via Componential Set-Based Analysis*. PhD thesis, Rice University, Houston, Texas, May 1997.
- [2] Cormac Flanagan and Matthias Felleisen. Set-based analysis for full Scheme and its use in soft-typing. Technical Report TR95-254, Rice University, 1995.
- [3] Cormac Flanagan and Matthias Felleisen. Modular and polymorphic set-based analysis: Theory and practice. Technical Report TR-96-266, Rice University, 1996.
- [4] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 235–248, June 1997.
- [5] Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Finding bugs in the web of program invariants. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 23–32, 1996.

Index

∴, 16

assertion, 16

debugger, static, 1, 2

define-constructor, 16

define-type, 16

MrSpidey, 1, 2

polymorphic, 16

polymorphism, 16

type, 14

type, assertion, 16

type, class, 14

type, empty, 14

type, list, 14

type, listof, 14

type, mu, 14

type, object, 14

type, rec, 14

type, union, 14

type∴, 16