

# Some Sparkle Documentation

Malcolm Dowse

March 2, 2004

## 1 Logic

Sparkle's logic is derived from a variant sequent calculus for natural deduction. Each step in a proof takes the form of a *sequent*  $A_1, A_2, \dots, A_n \vdash B$  which asserts that  $B$  is true if assumptions  $A_1, A_2, \dots, A_n$  are also true. For logical axioms, this is usually written  $\Gamma \vdash B$ , where  $\Gamma$  is a syntactic metavariable for finite sets of formulae. Set union is written with a comma and set brackets are left out:  $\Gamma, A_1$  is written instead of  $\Gamma \cup \{A_1\}$ .

Within Sparkle each assumption is explicitly named, usually in the form H1, H2 etc. The sequent  $\Gamma, \mathbf{Hn}: A \vdash B$  indicates that  $\mathbf{Hn}$  is the name of assumption  $A$ . Proofs are performed in a backwards style in Sparkle: beginning with the proof goal, one proceeds backwards towards a known tautology, or existing theorem. For this reason, all inference rules should be read from bottom to top.

All the standard logical operators are supported ( $\wedge, \vee, \implies, \iff, \neg, \forall, \exists$ ), and the boolean constants are **TRUE** and **FALSE**.

Each inference rule corresponds to a Sparkle tactic.

### 1.1 Conjunction

$$\begin{array}{l} \text{Split Deep Hn} \quad \frac{\Gamma, A, B \vdash C}{\Gamma, \mathbf{Hn}: A \wedge B \vdash C} \\ \\ \text{Split Deep} \quad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \end{array}$$

### 1.2 Disjunction

$$\text{Case Deep Hn} \quad \frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, \mathbf{Hn}: A \vee B \vdash C}$$

$$\text{Left} \quad \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}$$

$$\text{Right} \quad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}$$

### 1.3 Implication

$$\text{Introduce Hn} \quad \frac{\Gamma, \text{Hn}: A \vdash B}{\Gamma \vdash A \implies B}$$

$$\text{Cut Hn} \quad \frac{\Gamma, A \vdash A \implies B}{\Gamma, \text{Hn}: A \vdash B}$$

$$\text{Apply Hn} \quad \frac{\Gamma, A \implies B \vdash A}{\Gamma, \text{Hn}: A \implies B \vdash B}$$

$$\text{Apply Hn to Hm} \quad \frac{\Gamma, A, B \vdash C}{\Gamma, \text{Hm}: A, \text{Hn}: A \implies B \vdash C}$$

### 1.4 Negation

$$\text{Contradiction} \quad \frac{\Gamma, A \vdash \text{FALSE}}{\Gamma \vdash \neg A}$$

$$\text{Contradiction Hn} \quad \frac{\Gamma, \neg A \vdash A}{\Gamma, \text{Hn}: \neg A \vdash B}$$

### 1.5 Universal Quantification

$$\text{Introduce x} \quad \frac{\Gamma \vdash A}{\Gamma \vdash \forall x. A} \quad (\text{x not free in } \Gamma)$$

$$\text{Specialize Hn with t} \quad \frac{\Gamma, A[t/x] \vdash B}{\Gamma, \text{Hn}: \forall x. A \vdash B}$$

### 1.6 Existential Quantification

$$\text{Witness t} \quad \frac{\Gamma \vdash A[t/x]}{\Gamma \vdash \exists x. A}$$

$$\text{Witness for Hn} \quad \frac{\Gamma, A \vdash B}{\Gamma, \text{Hn}: \exists x. A \vdash B} \quad (\text{x not free in } \Gamma, B)$$

## 1.7 Logical Equivalence

$$\text{SplitIff} \quad \frac{\Gamma \vdash A \Longrightarrow B \quad \Gamma \vdash B \Longrightarrow A}{\Gamma \vdash A \Longleftrightarrow B}$$

$$\text{SplitIff Hn} \quad \frac{\Gamma, A \Longrightarrow B, B \Longrightarrow A \vdash C}{\Gamma, \text{Hn}: A \Longleftrightarrow B \vdash C}$$

## 1.8 Other logical rules

$$\text{Discard Hn} \quad \frac{\Gamma \vdash B}{\Gamma, \text{Hn}: A \vdash B}$$

$$\text{Assume (A)} \quad \frac{\Gamma, A \vdash B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

$$\text{Exact Hn} \quad \Gamma, \text{Hn}: A \vdash A$$

$$\text{Trivial} \quad \Gamma \vdash \text{TRUE}$$

$$\text{Absurd Hn Hm} \quad \Gamma, \text{Hn}: A, \text{Hm}: \neg A \vdash B$$

## 2 Core-Clean

Core-Clean is a stripped down version of the Clean programming language. All variables are lambda-lifted. Pattern matching and guards are converted into case-expressions. Unique and existential types are not permitted. The use of type-classes (ad-hoc polymorphism) is permitted, but probably inadvisable at the minute. Strictness annotation of function types is retained, as are algebraic data-types, let-expressions and parametric polymorphism.

One reasons about Core-Clean programs using equality:  $p1=p2$  states that the program  $p1$  has the exact same (extensional) behaviour as  $p2$ . That is: substituting the sub-program  $p1$  for  $p2$  in any program will not affect its return value (although it may affect its efficiency). The expression equality operator is reflexive, symmetric and transitive.

In Clean/Core-Clean it is possible that arbitrary programs contain undefined sub-expressions - sub-expressions that will never reduce to WHNF. This is an inescapable consequence of Turing completeness. Therefore, in order to reason formally as to whether two programs behave in the same way, it is necessary to *lift* all types so that  $\perp$  is an element. If an expression is of

type `Bool`, it may return `True`, it may return `False`, or it may never return at all ( $\perp$ ). This is the same with all types. An expression returning a (lazy) list of `Int` may have the following behaviour: `[12:[ $\perp$ : [3:[4: $\perp$ ]]]]`.

If `p1` and `p2` are Core-Clean programs/expressions, `p1=p2` is a logical proposition and may eventually be proven correct. Arbitrary Core-Clean expressions on their own are not logical propositions, but if the type of an expression `p1` is `Bool` (computational, three-valued truth), then asserting `p1` is short-hand for `p1=True`; that it behaves the same as a program which returns `True`.

### 3 Logical and Program Equivalence

Reflexive	$\Gamma \vdash A = A$
Reflexive	$\Gamma \vdash A \iff A$
Symmetric	$\frac{\Gamma \vdash A = B}{\Gamma \vdash B = A}$
Symmetric	$\frac{\Gamma \vdash A \iff B}{\Gamma \vdash B \iff A}$
Symmetric Hn	$\frac{\Gamma, A = B \vdash C}{\Gamma, \text{Hn}: B = A \vdash C}$
Symmetric Hn	$\frac{\Gamma, A \iff B \vdash C}{\Gamma, \text{Hn}: B \iff A \vdash C}$
Transitive B	$\frac{\Gamma \vdash A = B \quad \Gamma \vdash B = C}{\Gamma \vdash A = C}$
Transitive B	$\frac{\Gamma \vdash A \iff B \quad \Gamma \vdash B \iff C}{\Gamma \vdash A \iff C}$

### 4 Reduction

`Reduce` is probably the most important tactic in Sparkle. It evaluates a Core-Clean expression or sub-expression either step-by-step or in one go. There are four components in a single `Reduce` command.

- **Reduction Strategy** This determines the conditions by which any arbitrary sub-expression should be expanded. For example, it often makes no sense to expand the definition of the function  $f$  in expression  $(f (g a))$  until sub-expression  $(g a)$  has been expanded. There are three strategies:
  - **Reduce** “Normal”. Reduce as normal in Clean. Variables are treated as non-WHNF.
  - **Reduce-** “Defensive”. “As Normal, but if a function is ‘mathematically’ strict (meaning the body of the function will reduce to  $\perp$  when the argument is  $\perp$ ) and it has a variable on that argument position, it *will* expand. Sparkle deduces which arguments are mathematically strict; the results of this simple analysis can be viewed when showing a function definition.”
  - **Reduce+** “Offensive”. Always expand function if possible.
- **Step Count** The step count can either be a number, which states how many reduction steps should be applied, or it can be **NF** or **RNF**. If it’s the latter, it will reduce until the sub-expression is in normal-form or head-normal-form respectively. If none is specified, **NF** is chosen by default.
- **Redex Selection** This selects which sub-expression should be reduced, and can either be **All** or  $(fname\ n)$ . The first says to reduce all sub-expressions (ie. the whole expression), and the second selects the  $n$ ’th occurrence of function  $fname$ . If none is specified, **All** is chosen by default.
- **Hypothesis Selection** If it exists, this indicates which assumption should be reduced, or if none is specified the expression in the current proof-goal is reduced.

Some example reduction steps:

- **Reduce** Reduce all redexes in proof-goal to normal-form.
- **Reduce 1 (case 1)** Reduce the first case-expression in the proof-goal by one-step.
- **Reduce+ (fst 2) in H2** Offensively reduce the second occurrence of the **fst** function in assumption **H2**.
- **Reduce RNF H4** Reduce all redexes in assumption **H4** to head-normal-form.

The tactic `Opaque fn` will cause function `fn` to never be reduced – unless reversed by a `Transparent fn` tactic.

## 5 Rewrite

The `Rewrite` tactic takes an assumption that defines the equality of two expressions, and uses that equality in either the main proof-goal or another assumption.

`Rewrite -> H3` and `Rewrite <- H3` rewrite, either from left-to-right or right-to-left, all valid occurrences in the main proof-goal of the equality defined in assumption `H3`.

`Rewrite -> n H3` does the same except it just replaces the `n`'th occurrence.

`Rewrite -> H3 in H7` rewrites sub-expressions in the assumption `H7`, not the main proof-goal.

The `Rewrite` tactic can be quite smart. Even if the equality has pre-conditions and universally quantified variables, it will still work. Any pre-conditions will have to be proven as separate sub-goals.

## 6 Integers

`Ints` are special in `Core-Clean` and in `Sparkle`, and a number of tactics deal only with integer arithmetic and integer logic.

The tactic `IntArith` performs simple arithmetic on integers. It will convert an expression such as `(1+2)-3` into `0`, or `n+3+5` into `n+8`, but cannot reduce `(n+4)-n` into `4`.

`IntCompare` can prove some simple facts to do with integer equality and inequality, but nothing too complicated. Without extra theorems, inequalities usually have to be proven in a painful single-step manner.

The `Compare n1 with n2` tactic will (effectively) turn a sequent of the form  $\Gamma \vdash B$  into five sub-goals:

$$\begin{aligned} & \Gamma, n1 = \perp \vdash B \\ & \Gamma, n2 = \perp \vdash B \\ & \Gamma, n1 < n2 \vdash B \\ \Gamma, n1 \neq \perp, n2 \neq \perp, n1 = 0 & \vdash B \\ & \Gamma, n2 < n1 \vdash B \end{aligned}$$

## 7 Algebraic Data Types

Without going into a huge technical description, the `Cases` tactic performs case analysis on a particular algebraic data-type.

If expression `e` has an algebraic type with  $n$  data-constructors, the tactic `Cases e` forms  $n + 1$  of sub-goals: in each sub-goal, all occurrences of `e` are replaced by one specific data-constructor or  $\perp$ .

The `Cases` tactic can be explicit. `Explicit cases e` forms the same number of sub-goals, but doesn't do the replacement automatically. It instead introduces a new assumption defining what `e` is equal to.

The tactic `SplitCase n` (or `SplitCase n in Hn`) takes the  $n$ 'th case-expression (in assumption `Hn`) and creates a number of sub-goals depending on `e`'s type and the existence of a default clause in the case-expression.

Depending on some strictness properties, the `MoveInCase` tactic allows an expression on the outside of a case-expression to be moved into the result of each individual case. For example:

```
fst (case b of
  True  -> (3,5)
  False -> (4,7))
```

.. can be converted into ..

```
case b of
  True  -> fst (3,5)
  False -> fst (4,7)
```

The `AbsurdEquality` tactic asserts that two different data-constructors are indeed different. ie.  $\neg(\text{True} = \text{False})$ , or  $\neg(\text{Just } x = \text{Nothing})$ .

## 8 Definedness and Strictness

Sparkle is quite clever at detecting if a variable is  $\perp$ . If the text `(defined)` appears beside a particular variable `x`, that is short-hand for the knowledge that  $x \neq \perp$ . Similarly, if the text `(undefined)` appears, that means  $x = \perp$ .

If it can prove that `x` is both defined and undefined, the `Definedness` tactic will automatically prove the proof-goal by contradiction.

The `RefineUndefinedness` tactic is used to turn an assumption of the form `(f x y) ≠ ⊥` into something of the form  $x \neq \perp \vee y \neq \perp$ .

Sparkle is able to deduce a great deal about functions based on whether the left-hand-side of the function type has strictness annotation.

## 9 Induction

Induction can be performed on universally quantified integer variables or universally quantified variables of algebraic type with the `Induction v` tactic.

With integers, induction can always be performed, and will result in four sub-goals: the number is undefined, the number is negative, the number is zero, and the inductive case.

With algebraic types, the expression must be admissible (i.e. form a chain-complete domain). If it is,  $n + 1$  subgoals will result, one for each data-constructor and one for  $\perp$ . If it's not, or it is and it cannot be proven, Sparkle will complain.

An informal description of admissibility is that the truth of a proof-goal which contains possibly infinite data-structures can be ascertained from the truth of the same proof-goal where all structures are finite. If inducting over a certain structure is inadmissible, one solution is only reasoning about those structures which are finite. This is possible, in Sparkle, by carefully choosing an integer variable which acts as a measure for the size of the structure in question. So if inducting over `xs` in  $P(xs)$  is inadmissible, inducting over `n` in  $n = \text{length } xs \rightarrow P(xs)$  may still be useful.

## 10 Other tactics

There are a few miscellaneous tactics:

- `Rename n1 to n2` renames a variable or assumption.
- `Undo` undoes the last proof tactic - so it's more of a meta-tactic.
- `Uncurry` is sometimes necessary to turn an expression like  $((f\ a)\ b)$  into  $(f\ a\ b)$ .
- `Injective` attempts to prove an equality like  $f\ a1\ b1\ c1 = f\ a2\ b2\ c2$  by proving  $a1 = a2 \wedge b1 = b2 \wedge c1 = c2$ . This is particularly useful when `f` is a data-constructor.
- `Extensionality` allows one to prove  $f1 = f2$ , where `f1` and `f2` are functions, by proving that  $\forall x. f1\ x = f2\ x$ .

## 11 Proof language

From the Clean mailing-list, Monday 20 January 2003:

```

> Hi, All!
>
> Is there any manual on using Sparkle besides "Theorem proving for
> functional programmers" paper available? What is the syntax used for
> entering new theorems?

```

Unfortunately, there is no manual for Sparkle available. The only other very limited source of information is the website at <http://www.cs.kun.nl/Sparkle>.

The syntax for entering new theorems is as follows:

```

-----
EQUALITY:                "E = E"
NEGATION:                "~p"
CONJUNCTION:            "P /\ Q"
DISJUNCTION:            "P \/ Q"
IMPLICATION:            "P -> Q" or "P => Q"
EQUIVALENCE:            "P <-> Q" or "P <=> Q"
UNIVERSAL QUANTIFICATION: "[x::T] P"
EXISTENTIAL QUANTIFICATION: "{x::T} P"
-----

```

Notes:

- \* The types in the quantifications may be omitted. (Sparkle tries to derive types for variables automatically, but this might fail in some case, such as overloading).
- \* Universal quantifications may be omitted completely. (Sparkle automatically adds universal quantifications for all free variables it finds)

## 12 Command-Line Interface

There are three ways of applying tactics: The “List of tactics” window, the “Tactics Suggestion” window and the command-line interface.

Not only is the command-line often faster than the other two, it is a little more powerful than the other two since it allows for some primitive flow-control. To perform a tactic, enter it followed by a full-stop. Tactics can be sequenced with a `;`, and `Repeat (t)` will repeatedly perform `t` as many times as possible.

## 13 Known Bugs

The following bugs are currently known:

- There is no way to input the arithmetic negation operator  $\sim$ .
- Induction on integers can only be performed from the command-line.
- The `ExpandFun` tactic is badly broken and shouldn't be used.
- The proof-language often cannot parse something of the form  $((f\ a)\ b) = c$ . It is necessary to rewrite this as  $g = f\ a \rightarrow g\ b = c$ .
- Sparkle can detect cyclic dependencies between theorems (a logical error), but cannot detect cyclic dependencies between sections. Before saving sections it is important to check that if a theorem in section A depends on a theorem in section B, there doesn't exist a (different) theorem in section B that depends on a (different) theorem in section A. If this occurs, neither section will be loadable (at all).
- Pattern-matching on lists within let-expressions seems to be problematic.
- Sparkle has trouble type-checking algebraic data-types if the type-definition uses strictness-annotation or macros.
- The current `Extensionality` tactic is a little too powerful. It should also create a sub-goal stating that the functions are either both undefined, or both defined.
- Lambda-abstractions should be used with caution. All lambda-abstractions are turned into named functions by the Clean compiler (given names like `\;441;32`), and these names may vary if the ICL file is changed. If these names are ever used explicitly in a proof, it is very possible that the proof will fail to load correctly at a later date.
- When browsing a proof with the khaki-coloured modal window, pressing the Undo button rolls the proof back to the point at which the Browse button was originally clicked, not to the point displayed. In other words, the Previous and Next buttons have no effect on the result of the Undo button.

## 14 Installation Issues

### 14.1 Versions

The most up-to-date version of Clean is version 2.1.0, available directly from the <http://www.cs.kun.nl/~clean>. This includes Sparkle 0.0.4.

Unfortunately, there exists a memory-leak in this version of Sparkle, causing it to be about 10 times slower than normal. Therefore, I've been using Clean 2.0.2, with Sparkle 0.3a. The Sparkle source code is no different, except it has been compiled with the 2.0.2 compiler.

It is available from <http://www.cs.kun.nl/~clean/download/Clean20/windows>.

### 14.2 Using SEC files

Before Sparkle can read section files they must be copied into the `Clean 2.x/Tools/Sparkle 0.x/Sections` directory.