This is a brief report on the changes made to the (Penn) Lambda Calculator by Dylan Bumford at New York University between 05-2013 and 09-2013. Development remains active.

Interface

For all users, the program now opens immediately to the TrainingWindow, since this is almost always what is desired. The ScratchPad and TeacherTool are available from the menu bar.

For Mac users, the program is now bundled as an Apple application. In the interest of user experience, this means that the menu options now appear in the native Mac menu bar at the top of the screen. Shortcut accelerator keys are relativized to platform (Control on Windows, Command on Mac). The application, dock, and Command-Tab icons now display the large red-circled lambda in logo-large.icns (or logo.ico for Windows). The menu includes an "About" item to display the version, developers, and affiliations. "Quitting" the application via the menu item or the Command-Q shortcut now fully shuts down the program, instead of merely closing the main window, which is instead accomplished by a menu item in "File", or the usual Command-W.

The "Latex" button now exposes a pretty-printed "qtree" version of the current node to the user. The tree is embedded in a minimal LaTeX document, and displayed as plain, copyable text in the NodePanel where feedback is given.

Indices

The notation for indices and types is now flexible. The index of a typed element may now follow or precede the type designation:

- fine: `t<e,t>_1`
- fine: `t_1<et,t>`

Abstraction indices now inherit the types of the traces they are coindexed with, unless they are themselves explicitly typed. If the abstraction index is coindexed with multiple traces, and the traces do not have the same type, the abstraction index inherits the type of the the most deeply embedded trace. This may or may not be a good idea, but then again, leaving multiple traces with multiple types may or may not be a good idea to begin with.

Intensional Function Application

Intensional Function Application should be available as a fully-functioning composition rule, along with FA, PM, and the others. In particular, the bugs pointed out by Mats Rooth causing issues with the typing conventions were caught.

Polymorphism

Atomic types now come in two varieties: concrete and variable. Concrete types are designated by the familiar `e`, `t`, `s`, `n`, `v`, and `i`, and behave as they used to. All other atomic types (e.g. `a`, `b`, etc.) are considered variable.

Atomic variable types "equal" to all other types. Product types and composite types are compared component-wise. For example:

- `<a> == <e>`
- `<a> == <b>`
- `<a> == <et>`
- `<at> == <et>`
- `<at> == <et,t>`
- `<at> != <e,tt>`

- `<at> != <e>`

This means that for purposes of Function Application, for instance, a function of type `<at>` will accept an individual argument of type `e` and produce a truth value of type `t` as a result.

More generally, when a flexible function encounters a concrete argument, all of the type variables in its signuture are made concrete. For example, when the fully general function composition operator, of type `<<bc>,<<ab>,<ac>>>` is applied to a specific argument of type `<et>`, the remaining function composition closure is reset to type `<<ae>,<at>>`. That is, while type `a` is still free, the type variables `b` and `c` have been "captured" by the `et` property that saturated the first argument. At the next application step, another concrete argument will fix the type of `a`, and thus by the third step the final closure will be entirely concrete.

Flexible functions (often called combinators) applied to concrete lexical items is a typical use case for polymorphism in semantics. But perhaps from time to time, it can come in handy for combinators to apply to other combinators, or for some lexical (e.g. coordinating conjunctions, pronouns) to take multiple types of arguments. This is supported. For instance, the Predicate Modification rule will now combine any two arguments of the form `<xt>`, where `x` is any contant, variable, or composite type. We might give `and` an object language denotation along the same lines, of type `<<at>,<<at>,<at>>>`. The first argument (if it serves a concrete domain) will fix the required type of the second and third arguments.

This flexibility comes at a cost. A nonterminal node dominating two children, both of type `<at>`, will no longer satisfy the demands of at most one composition rule. In fact, it will satisfy three: Predicate Modification, Left-to-Right Function Application, and Right-to-Left Function Application. Which of these rules, if any, are actually appropriate depends on the semantics of the arguments and the arguments of the semanticists. For a (rather contrived) example, we might imagine that the property "good" may be equally well ascribed to individuals ("this book is good") and properties ("reading is good"), in which case it's type would have to that of the polymorphic set: `<at>`. But then when it is used attributively ("this is a good book"), the calculator will have to decide whether "good" should be applied to its argument or conjoined with its argument. The former results in a truth value, the latter in a property. If we are running in student-mode, the calculator is a student, and all is well. The student will have to choose between competing applicable composition rules and live with his or her choices. If in God-mode, the calculator is a machine, and things are not well. At the moment, rather than giving up on the node, the program presents the instructor with a dialog asking which of the composition rules he or she would like before attempting to represent the exercise.

Aside from this isolated request for divine intervention, type-inference errors are still essentially handled at runtime. That is, type imcompatibilities lead to nonterminal nodes with no available composition rules. The error is not passed to the user until the nodes are evaluated, at which point a "Problem" type is generated and a feedback message displayed.

A Few Potential Improvements

At the moment the distinction between constant atomic types and variable atomic types is hardcoded into the typing conventions. This could be made more systematic if the `AtomicType` class were generalized to accept strings as type identifiers rather than characters. In this case, we might follow an ML-style convention of marking type variable identifiers with primes.

There is currently no way to restrict a type signature to a subset of conceivable concrete instantiations. Any given atomic component of a type is either locked down, or fully permiscuous, but in practice, one may wish to extend type flexibility somewhat while preserving some degree of safety. For instance, Predicate Modification is not quite general enough, as things stand, since it only applies to types that end abruptly in `t`. Really it should apply to any type that ends in a "conjoinable type" (`t` or `<at>` or `<b,<at>>` or `<c,<b,<at>>>`, etc.). Building this sort of restriction into Predicate Modification itself is not difficult, but building it into a user-defined entry for "and" is currently impossible.

The LaTeX view is sufficient but inelegant. A dedicated window with syntax-highlighted output would be a nice touch. It should also be easy to add further bells and whistles to the generated trees, including automatic movement arrows connecting quantifers and their traces.