Equational Reasoning for Linking with First-Class Primitive Modules

Joe Wells and René Vestergaard
Heriot-Watt University
Computing & Electrical Engineering Department
http://www.cee.hw.ac.uk/~jbw
http://www.cee.hw.ac.uk/~jrvest

September 20, 2000

Overview

- Motivation.
- The m-Calculus.
- An m-Calculus Toolkit.
- Module Systems via m-Calculus.
- Confluence.
- Related/Future Work and Conclusion.

Some Desirable Goals for a Module Formalism

- Reuse (linked in different ways) without source code modification or access. "Black-box reuse".
- Intra-language modules and linking.
- First-class modules.
- Formalism should be reasonably close to real software practice. It should support such things as mutual dependencies and hierarchical namespaces.
- Formalism should have well defined semantics and support for showing correctness of transformations.

Existing Practice: C

- The "module system" of C extensively involves features that are not part of the C language. Linking is specified with makefiles. Generic modules are frequently implemented via some preprocessor.

- Namespace of modules is the filesystem. All components exported from modules appear in a single top-level C-language namespace. Inter-module connections are hard-wired to component names. There is no support for renaming components.

- C modules do not support black-box reuse. There is limited (extra-linguistic) support for first-class modules. A hierarchical namespace is not directly supported by C modules.

Existing Practice: Packages

- Languages like Ada, Modula-3, and Haskell have a kind of module we call *packages*.

- Two-layer namespace — a top level of modules and within each module a flat layer of component names. Inter-module connections are hard-wired to module/component name pairs.

- Packages do not support black-box reuse or a hierarchical namespace. Packages are not first-class.

Existing Practice: SML

- Module namespace is fully hierarchical.

- Two kinds of inter-module connections:
  - There is a linear order of module declarations. Later modules can use components of earlier ones by name.
  - A *functor* abstracts a module (a *structure*) over references to other modules and can then be applied to different argument modules later.

- ML supports limited black-box reuse via functors.

- Unlike other module systems, mutually dependent modules are forbidden. Both kinds of connections are asymmetric.

A Generalization: Incomplete Modules and Linking

$M_1 =$

```
(module
  exported f = N(f,g,i)
  deferred g
  deferred h
  private i = O(h))
```

$M_2 =$

```
(module
  deferred f
  exported g = P(f,i)
  deferred h
  private i = Q)
```

$M_1 \oplus M_2 =$

```
(module
  exported f = N(f,g,i)
  exported g = P(f,i')
  deferred h
  private i = O(h)
  private i' = Q)
```

Overview

- Motivation.
- The m-Calculus.
- An m-Calculus Toolkit.
- Module Systems via m-Calculus.
- Confluence.
- Related/Future Work and Conclusion.

The m-Calculus: Syntax

- The main constructor is the *module* (more on next page):

$$\{F_1 \triangleright x_1 = B_1,\ \ldots,\ F_n \triangleright x_n = B_n\}$$

- The operations are *linking*, *selection*, *hiding*, and *sieving*:

$$M_1 \oplus M_2, \quad M_1.f, \quad M\backslash f, \quad M \backslash\!- \{f_1, \ldots, f_n\}$$

- Although it is essentially definable, for convenience the auxiliary *letrec* constructor is included:

$$\langle M \mid x_1 = M_1,\ \ldots,\ x_n = M_1\rangle$$

Terminology for m-Calculus Modules

external names *empty body* *anonymous marker*

$$\{\underbrace{\underbrace{\mathrm{f} \triangleright \mathrm{x} = \mathrm{M},\ \mathrm{g} \triangleright \_ = \mathrm{N},}_{\textit{exported}}\ \underbrace{\mathrm{h} \triangleright \mathrm{y} = \bullet,}_{\textit{deferred}}\ \underbrace{\_ \triangleright \mathrm{z} = \mathrm{O}}_{\textit{private}}}_{\textit{components}}\}$$

- The *internal* name "_" abbreviates a variable occurring nowhere else.
- A module has features of a λ-abstraction, a record, and an environment.

The m-Calculus: Semantics

- The first major operation is *linking*:

$$\{\mathrm{f} \triangleright \mathrm{x} = \mathrm{M},\ \mathrm{g} \triangleright \mathrm{z} = \mathrm{N},\ \mathrm{h} \triangleright \mathrm{y} = \bullet\} \oplus \{\mathrm{g} \triangleright \mathrm{z} = \bullet,\ \mathrm{h} \triangleright \mathrm{y} = \mathrm{O},\ \_ \triangleright \mathrm{w} = \mathrm{P}\}$$

$$\downarrow\ (\textbf{link})$$

$$\{\mathrm{f} \triangleright \mathrm{x} = \mathrm{M},\ \mathrm{g} \triangleright \mathrm{z} = \mathrm{N},\ \mathrm{h} \triangleright \mathrm{y} = \mathrm{O},\ \_ \triangleright \mathrm{w} = \mathrm{P}\}$$

- The second major operation is *selection*:

$$\{\mathrm{f} \triangleright \mathrm{x} = \mathrm{M},\ \mathrm{g} \triangleright \mathrm{z} = \mathrm{N},\ \mathrm{h} \triangleright \mathrm{y} = \mathrm{O},\ \_ \triangleright \mathrm{w} = \mathrm{P}\}.\mathrm{g}$$

$$\downarrow\ (\textbf{select})$$

$$\langle \mathrm{z} \mid \mathrm{x} = \mathrm{M},\ \mathrm{z} = \mathrm{N},\ \mathrm{y} = \mathrm{O},\ \mathrm{w} = \mathrm{P}\rangle$$

Substitution in the m-Calculus

- The m-calculus does substitution *one* target at a time:

$$\{\mathrm{f} \triangleright \mathrm{x} = \mathrm{M},\ \mathrm{g} \triangleright \_ = \underline{\mathrm{x}}\} \longrightarrow \{\mathrm{f} \triangleright \mathrm{x} = \mathrm{M},\ \mathrm{g} \triangleright \_ = \mathrm{M}\}$$
$$\langle \underline{\mathrm{x}} \mid \mathrm{x} = \mathrm{M},\ \mathrm{y} = \mathrm{N} \rangle \longrightarrow \langle \mathrm{M} \mid \mathrm{x} = \mathrm{M},\ \mathrm{y} = \mathrm{N} \rangle$$

  Garbage collection rules later remove unreferenced bindings.

- Cyclic bindings must sometimes be kept. Keeping bindings in the same place *and* substituting for all occurrences is just less flexible. Bindings can be removed from their current location by including copies with replacements [Ancona and Zucca, 1999], but this loses sharing and explodes size, e.g.:

$$\begin{aligned} &\langle (\mathrm{x},\mathrm{x}) \mid \mathrm{x} = (\mathrm{x},\mathrm{y}),\ \mathrm{y} = (\mathrm{y},\mathrm{x}) \rangle \\ &\quad \longrightarrow (\langle \mathrm{x} \mid \mathrm{x} = (\mathrm{x},\mathrm{y}),\ \mathrm{y} = (\mathrm{y},\mathrm{x}) \rangle, \langle \mathrm{x} \mid \mathrm{x} = (\mathrm{x},\mathrm{y}),\ \mathrm{y} = (\mathrm{y},\mathrm{x}) \rangle) \end{aligned}$$

Acyclic Substitution Restriction

- The m-calculus does not perform substitution along cycles:

$$\{\mathrm{f} \triangleright \mathrm{x} = (1, \mathrm{y}),\ \mathrm{g} \triangleright \mathrm{y} = (\underline{\mathrm{x}}, 2)\} \not\longrightarrow \{\mathrm{f} \triangleright \mathrm{x} = (1, \mathrm{y}),\ \mathrm{g} \triangleright \mathrm{y} = ((1, \mathrm{y}), 2)\}$$

- The restriction is needed for the m-calculus to be confluent. Ariola and Klop [1997] give an example showing that confluence is impossible without the restriction.

- Because only substitution into letrec bodies is needed for evaluation, this restriction is merely a loss in equational reasoning power.

- If this restriction were removed, we believe the m-calculus would still be consistent, but different proof methods would be needed to show this.

Dynamic Errors in the m-Calculus

- Linking modules with common output components:

$$\{\mathrm{f} \triangleright \mathrm{w} = \bullet,\ \mathrm{g} \triangleright \mathrm{x} = \mathrm{M}\} \oplus \{\mathrm{f} \triangleright \mathrm{y} = \mathrm{N},\ \mathrm{g} \triangleright \mathrm{z} = \mathrm{N}'\}$$

- Selecting from an incomplete module:

$$\{\mathrm{f} \triangleright \mathrm{w} = \bullet,\ \mathrm{g} \triangleright \mathrm{x} = \mathrm{M}\}.\mathrm{g}$$

- Selecting a nonexistent component:

$$\{\mathrm{f} \triangleright \mathrm{w} = \mathrm{M},\ \mathrm{g} \triangleright \mathrm{x} = \mathrm{N}\}.\mathrm{h}$$

- Hiding an input component:

$$\{\mathrm{f} \triangleright \mathrm{w} = \bullet,\ \mathrm{g} \triangleright \mathrm{x} = \mathrm{N}\}\backslash\mathrm{f}$$
$$\{\mathrm{f} \triangleright \mathrm{w} = \bullet,\ \mathrm{g} \triangleright \mathrm{x} = \mathrm{N}\}\backslash{-}\{\mathrm{g}\}$$

Overview

- Motivation.
- The m-Calculus.
- An m-Calculus Toolkit.
- Module Systems via m-Calculus.
- Confluence.
- Related/Future Work and Conclusion.

λ-Calculus in m-Calculus

- The λ-calculus can be encoded by choosing two distinguished component names "arg" and "res":

$$(\lambda x.M) = \{\mathrm{arg} \triangleright x = \bullet,\ \mathrm{res} \triangleright \_ = M\}$$

$$(M M') = (M \oplus \{\mathrm{arg} \triangleright \_ = M'\}).\mathrm{res}$$

  (Example of rewriting simulation of β rule on next page.)

- The encoding is like that independently developed by Ancona and Zucca [1999]. It is only superficially like the encoding of λ-calculus in ς-calculus by Abadi and Cardelli [1996].

How the λ-Calculus Encoding Works

$$
\begin{array}{lrl}
 & & (\lambda x.x\,x)(y\,z) \\
 & = & (\{\mathrm{arg} \triangleright x = \bullet,\ \ \mathrm{res} \triangleright w = (x\,x)\} \oplus \{\mathrm{arg} \triangleright x = (y\,z)\}).\mathrm{res} \\
(\textbf{link}) & \longrightarrow & \{\mathrm{arg} \triangleright x = (y\,z),\ \ \mathrm{res} \triangleright w = (x\,x)\}.\mathrm{res} \\
(\textbf{select}) & \longrightarrow & \langle w \mid x = (y\,z),\ \ w = (x\,x)\rangle \\
(\textbf{subst-letrec}) & \longrightarrow & \langle (x\,x) \mid x = (y\,z),\ \ w = (x\,x)\rangle \\
(\textbf{gc-letrec}) & \longrightarrow & \langle (x\,x) \mid x = (y\,z)\rangle \\
(\textbf{subst-letrec}) & \longrightarrow & \langle (y\,z)\,x \mid x = (y\,z)\rangle \\
(\textbf{subst-letrec}) & \longrightarrow & \langle (y\,z)\,(y\,z) \mid x = (y\,z)\rangle \\
(\textbf{gc-letrec}) & \longrightarrow & \langle (y\,z)\,(y\,z) \mid \ \rangle \\
(\textbf{empty-letrec}) & \longrightarrow & (y\,z)\,(y\,z)
\end{array}
$$

Records in m-Calculus

- Records are encoded as modules with components whose internal names are not used:

$$\{f_1 = M_1,\ \ldots,\ f_n = M_n\} \quad = \quad \{f_1 \triangleright \_ = M_1,\ \ldots,\ f_n \triangleright \_ = M_n\}$$

- The expected behavior is obtained:

$$\begin{array}{lll} & & \{\mathrm{f} = 1,\ \mathrm{g} = \mathrm{x}\}.\mathrm{g} \\ & = & \{\mathrm{f} \triangleright \mathrm{y} = 1,\ \mathrm{g} \triangleright \mathrm{z} = \mathrm{x}\}.\mathrm{g} \\ (\textbf{select}) & \longrightarrow & \langle \mathrm{z} \mid \mathrm{y} = 1,\ \mathrm{z} = \mathrm{x}\rangle \\ (\textbf{subst-letrec}) & \longrightarrow & \langle \mathrm{x} \mid \mathrm{y} = 1,\ \mathrm{z} = \mathrm{x}\rangle \\ (\textbf{gc-letrec}) & \longrightarrow & \langle \mathrm{x} \mid \ \rangle \\ (\textbf{empty-letrec}) & \longrightarrow & \mathrm{x} \end{array}$$

- Henceforth we let "$f = M$" abbreviate "$f \triangleright \_ = M$".

Other Encodings

The long version [Wells and Vestergaard, 1999] shows how to encode the following in the m-calculus:

- A component renaming operator that can rename both input and output components.
- The ς-calculus of objects of Abadi and Cardelli [1996].
- The module calculus of Ancona and Zucca [1999].
- The calculus of first-class contexts of Hashimoto and Ohori [1997].
- The $\lambda\varepsilon$-calculus of first-class environments of Sato et al. [1999].

Overview

- Motivation.
- The m-Calculus.
- An m-Calculus Toolkit.
- Module Systems via m-Calculus.
- Confluence.
- Related/Future Work and Conclusion.

C-style Modules

- Each object file O exporting components (functions, variables) $f_1, \ldots, f_n$, depending on imported components $g_1, \ldots, g_m$, and with private components $h_1, \ldots, h_o$ can be represented as a module M:

$$M = \{f_1 \triangleright x_1 = M_1,\ \ldots,\ g_1 \triangleright y_1 = \bullet,\ \ldots,\ \_ \triangleright h_1 = N_1,\ \ldots\}$$

- The linking of modules $M_1, \ldots, M_n$ to form a program and invoke its start routine is represented as:

$$P = (M_1 \oplus \ldots \oplus M_n).\mathrm{main}$$

- We are ignoring the call-by-value and imperative features of C-like languages here.

Package-style Modules

This example Haskell program:

```
module A (f) where
 f = N

module B (f, g) where
 import A
 g = P(A.f)
```

```
module Main (main) where
 import qualified B
 f = 5
 main = Q(B.f,B.g,f)
```

can be encoded as these m-calculus modules:

$$
\begin{aligned}
M_{A} &= \{A = \{f = N\}\} \\
M_{B} &= \{A \triangleright x = \bullet,\ B = \{g = P(x.f),\ f = x.f\}\} \\
M_{Main} &= \{B \triangleright x = \bullet,\ Main = \{\_ \triangleright y = 5,\ main = Q(x.f, x.g, y)\}\}
\end{aligned}
$$

Package-style Modules (more)

The previous example's modules were encoded as:

$$
\begin{array}{ll}
\mathrm{M}_{\mathrm{A}} & = \{\mathrm{A} = \{\mathrm{f} = \mathrm{N}\}\} \\
\mathrm{M}_{\mathrm{B}} & = \{\mathrm{A} \triangleright \mathrm{x} = \bullet,\ \mathrm{B} = \{\mathrm{g} = \mathrm{P}(\mathrm{x.f}),\ \mathrm{f} = \mathrm{x.f}\}\} \\
\mathrm{M}_{\mathrm{Main}} & = \{\mathrm{B} \triangleright \mathrm{x} = \bullet,\ \mathrm{Main} = \{\_ \triangleright \mathrm{y} = 5,\ \mathrm{main} = \mathrm{Q}(\mathrm{x.f}, \mathrm{x.g}, \mathrm{y})\}\}
\end{array}
$$

The evaluation of the program proceeds like this:

$$
\begin{array}{ll}
 & (\mathrm{M}_{\mathrm{A}} \oplus \mathrm{M}_{\mathrm{B}} \oplus \mathrm{M}_{\mathrm{Main}}).\mathrm{Main.main} \\
\longrightarrow\!\!\!\!\rightarrow & \left\{\begin{array}{l} \mathrm{A} \triangleright \mathrm{x} = \{\mathrm{f} = \mathrm{N}\},\ \mathrm{B} \triangleright \mathrm{z} = \{\mathrm{g} = \mathrm{P}(\mathrm{x.f}),\ \mathrm{f} = \mathrm{x.f}\}, \\ \mathrm{Main} = \{\_ \triangleright \mathrm{y} = 5,\ \mathrm{main} = \mathrm{Q}(\mathrm{z.f}, \mathrm{z.g}, \mathrm{y})\} \end{array}\right\}.\mathrm{Main.main} \\
\longrightarrow\!\!\!\!\rightarrow & \left\{\begin{array}{l} \mathrm{A} = \{\mathrm{f} = \mathrm{N}\},\ \mathrm{B} = \{\mathrm{g} = \mathrm{P}(\mathrm{N}),\ \mathrm{f} = \mathrm{N}\}, \\ \mathrm{Main} = \{\mathrm{main} = \mathrm{Q}(\mathrm{N}, \mathrm{P}(\mathrm{N}), 5)\} \end{array}\right\}.\mathrm{Main.main} \\
\longrightarrow\!\!\!\!\rightarrow & \mathrm{Q}(\mathrm{N}, \mathrm{P}(\mathrm{N}), 5)
\end{array}
$$

Handling Other Haskell Features

- Although the example used qualified names (e.g., "`B.f`"), nearly all of Haskell's behavior with unqualified names can be encoded.

- There is one strange quirk of Haskell's qualified names that would need an m-calculus extension to be handled: the "diamond import" behavior.

- Other Haskell module namespace management features can be handled, e.g., listing entities *not* to import with unqualified names, local aliases for imported modules, etc.

- Haskell's type classes should not be hard if one insists on working only with the result of the dictionary-passing translation. And no one knows how to do rewriting directly with type classes, so we don't feel bad about it.

SML-style Modules

- We are ignoring the type-level aspects of SML modules as well as SML's imperative features and call-by-value behavior.

- SML provides modules called *structures* as well as a λ-calculus (*functors* and *functor applications*) for manipulating them.

- SML structures can be encoded directly as m-calculus modules.

- SML functors and functor applications can be encoded using the λ-calculus encoding given earlier.

Overview

- Motivation.
- The m-Calculus.
- An m-Calculus Toolkit.
- Module Systems via m-Calculus.
- Confluence.
- Related/Future Work and Conclusion.

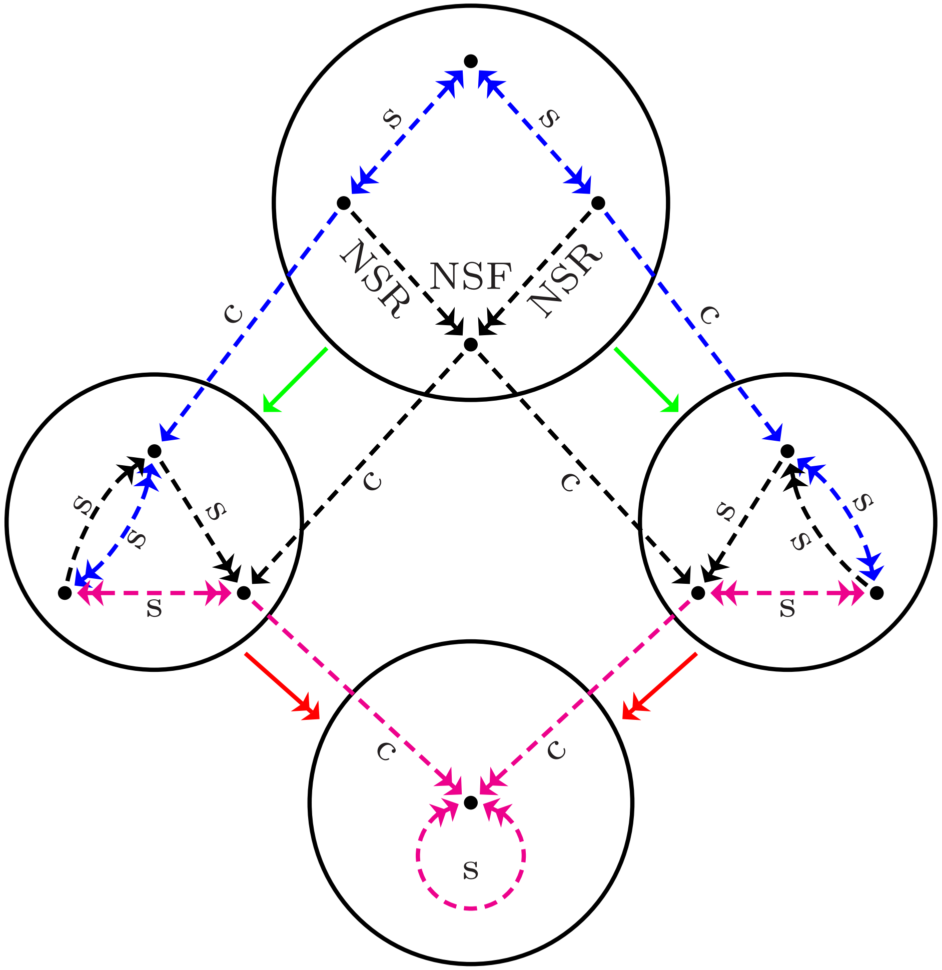
Confluence via Finite Developments

- Confluence of the m-calculus is shown via the well known method of *strong finite developments*, which means that any two coinitial developments can be joined by two further cofinal developments.

- Strong finite developments follows from *termination* and *weak confluence* of *marked rewriting*:
  - Termination of marked rewriting (a.k.a. finite developments) is proven using well known methods of weighted labels on terms.
  - Weak confluence of marked rewriting is proven as described on the following slide.

- The full proof can be found in [Wells and Vestergaard, 1999].

Weak Confluence of Marked Rewriting



○	(real) marked term
•	raw marked term
$\longrightarrow$	one step
$\twoheadrightarrow$	zero or more steps
$\twoheadleftarrow\!\!\twoheadrightarrow$	zero/more steps either way
——	on (real) marked terms
- - -	on raw marked terms
c	via computational rule(s)
s	via structural rule(s)
NSR	Nice Structural Rewriting
NSF	Nice Structural Form
——	premises
——	desired conclusion
- - -	equivalent to premises
- - -	equivalent to conclusion

Interesting Points about Proof Method

- Fully rigorous treatment of *structural* rewriting, including α-*conversion*.
- Nice structural term properties (e.g., "distinctly named bound variables") as side conditions on rules, *without* "Abracadabra!" style proofs.
- Convenient, like the convention in Barendregt [1984], but *rigorous*.
- The substitution on raw terms actually used is defined purely by *structural recursion* (as are similar term operations), making rigorous proofs much easier.

Overview

- Motivation.
- The m-Calculus.
- An m-Calculus Toolkit.
- Module Systems via m-Calculus.
- Confluence.
- Related/Future Work and Conclusion.

Related Calculi with Modules and Linking

	Rewrite Semantics	1st-Class Modules	1-Target Subst.	Confluence	Typed	General Comp.
Cardelli [1997]	yes	no	no	yes	yes	no
Ancona and Zucca [1998]	yes?	no	no	yes	no	no
Ancona and Zucca [1999]	yes	yes	no	yes?	yes	yes
Machkasova and Turbak [2000]	yes	no	yes	no	no	yes
Wells and Vestergaard [2000]	yes	yes	yes	yes	no	yes
Flatt and Felleisen [1998]	no	yes	n/a	n/a	no	yes
Drossopoulou et al. [1999]	no	?	n/a	n/a	yes	?

Future Work

- Type inference. The best candidate type system seems to be one by Harper and Pierce [1991] for polymorphic record concatenation.
- Extending calculus with features such as call-by-value evaluation, assignment statements, etc.
- Formalizing proof method in Coq.
- Using the design ideas of the m-calculus to make a more powerful module system for Haskell.

Conclusion

The m-calculus:

- Supports smooth encodings of the λ-calculus, renaming, records, first-class environments, etc.
- Supports first-class modules, mutual intermodule dependencies, hierarchical namespaces, and black-box reuse.
- Can represent several existing styles of module system.
- Has an extremely well behaved rewrite relation.

References

Martìn Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

Davide Ancona and Elena Zucca. An algebra of mixin modules. In F. Parisi Presicce, editor, *Recent Trends in Algebraic Development Techniques (12th Int'l Workshop, WADT '97 — Selected Papers)*, volume 1376 of *LNCS*, pages 92–106. Springer-Verlag, 1998.

Davide Ancona and Elena Zucca. A primitive calculus for module systems. In G. Nadathur, editor, *Proc. Int'l Conf. on Principles and Practice of Declarative Programming*, LNCS, Paris, France, 29 September – 1 October 1999. Springer-Verlag.

Zena M. Ariola and Jan Willem Klop. Lambda calculus with explicit recursion. *Inf. & Comput.*, 139:154–233, 1997.

Hendrik Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984.

Luca Cardelli. Program fragments, linking, and modularization. In *Conf. Rec. POPL '97: 24th ACM Symp. Princ. of Prog. Langs.*, 1997.

Sophia Drossopoulou, Susan Eisenbach, and David Wragg. A fragment calculus — towards a model of separate compilation, linking and binary compatibility. In *Proc. 14th Ann. IEEE Symp. Logic in Computer Sci.*, July 1999.

ESOP '00. *Proc. European Symp. on Programming*, volume 1782 of *LNCS*, 2000. Springer-Verlag.

Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Proc. ACM SIGPLAN '98 Conf. Prog. Lang. Design & Impl.*, pages 236–248, 1998.

Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. Technical Report CMU-CS-90-157R, Carnegie Mellon Univ., 2 July 1991.

Masatomo Hashimoto and Atsushi Ohori. A typed context calculus. Technical Report RIMS-1098, Research Inst. for Math. Sci., Kyoto University, 25 September 1997.

Elena Machkasova and Franklyn Turbak. A calculus for link-time compilation. In ESOP '00 ESOP '00, pages 260–274.

Masahiko Sato, Takafumi Sakurai, and Rod Burstall. Explicit environments. In

Fourth Int'l Conf. Typed Lambda Calculi and Applications, L'Aquila, Italy, 7–9 April 1999.

J. B. Wells and René Vestergaard. Confluent equational reasoning for linking with first-class primitive modules (long version). A short version is Wells and Vestergaard [2000]. Full paper with three appendices for proofs, August 1999.

J. B. Wells and René Vestergaard. Equational reasoning for linking with first-class primitive modules. In ESOP '00 ESOP '00, pages 412–428. A long version is Wells and Vestergaard [1999].