The History of Standard ML

DAVID MACQUEEN, University of Chicago, USA
ROBERT HARPER, Carnegie Mellon University, USA
JOHN REPPY, University of Chicago, USA

Shepherd: Kim Bruce, Pomona College, USA

The ML family of strict functional languages, which includes F#, OCaml, and Standard ML, evolved from the *Meta Language* of the LCF theorem proving system developed by Robin Milner and his research group at the University of Edinburgh in the 1970s. This paper focuses on the history of Standard ML, which plays a central rôle in this family of languages, as it was the first to include the complete set of features that we now associate with the name "ML" (i.e., polymorphic type inference, datatypes with pattern matching, modules, exceptions, and mutable state).

Standard ML, and the ML family of languages, have had enormous influence on the world of programming language design and theory. ML is the foremost exemplar of a functional programming language with strict evaluation (call-by-value) and static typing. The use of parametric polymorphism in its type system, together with the automatic inference of such types, has influenced a wide variety of modern languages (where polymorphism is often referred to as *generics*). It has popularized the idea of datatypes with associated case analysis by pattern matching. The module system of Standard ML extends the notion of type-level parameterization to large-scale programming with the notion of parametric modules, or *functors*.

Standard ML also set a precedent by being a language whose design included a formal definition with an associated metatheory of mathematical proofs (such as soundness of the type system). A formal definition was one of the explicit goals from the beginning of the project. While some previous languages had rigorous definitions, these definitions were not integral to the design process, and the formal part was limited to the language syntax and possibly dynamic semantics or static semantics, but not both.

The paper covers the early history of ML, the subsequent efforts to define a *standard* ML language, and the development of its major features and its formal definition. We also review the impact that the language had on programming-language research.

CCS Concepts: • **Software and its engineering → Functional languages**; **Formal language definitions**; **Polymorphism**; **Data types and structures**; **Modules / packages**; **Abstract data types**.

Additional Key Words and Phrases: Standard ML, Language design, Operational semantics, Type checking

ACM Reference Format:
David MacQueen, Robert Harper, and John Reppy. 2020. The History of Standard ML. *Proc. ACM Program. Lang.* 4, HOPL, Article 86 (June 2020), 100 pages. https://doi.org/10.1145/3386336

Authors' addresses: David MacQueen, Computer Science, University of Chicago, 5730 S. Ellis Avenue, Chicago, IL, 60637, USA, dmacqueen@mac.com; Robert Harper, Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA, 15213, USA, rwh@cs.cmu.edu; John Reppy, Computer Science, University of Chicago, 5730 S. Ellis Avenue, Chicago, IL, 60637, USA, jhr@cs.uchicago.edu.

2475-1421/2020/6-ART86
https://doi.org/10.1145/3386336

Contents

1 INTRODUCTION: WHY IS STANDARD ML IMPORTANT?

The ML family of languages started with the *meta language* (hence ML) of the LCF theorem proving system developed by Robin Milner and his research group at Edinburgh University from 1973 through 1978 [Gordon et al. 1978b].[1] Its design was based on experience with an earlier version of the LCF system that Milner had worked on at Stanford University [Milner 1972] .

Here is how Milner succinctly describes ML, the LCF metalanguage, in the introduction to the book *Edinburgh LCF* [Gordon et al. 1979]:

> . . . ML is a general purpose programming language. It is derived in different aspects from ISWIM, POP2 and GEDANKEN, and contains perhaps two new features. First, it has an escape and escape trapping mechanism, well-adapted to programming strategies which may be (in fact usually are) inapplicable to certain goals. Second, it has a polymorphic type discipline which combines the flexibility of programming in a typeless language with the security of compile-time type checking (as in other languages, you may also define your own types, which may be abstract and/or recursive); this is what ensures that a well-typed program cannot perform faulty proofs.

This original ML was an embedded language within the interactive theorem proving system LCF, serving as a logically-secure scripting language. Its type system enforced the logical validity of proved theorems and it enabled the partial automation of the proof process through the definition of proof *tactics*. It also served as the interactive command language of LCF.

The basic framework of the language design was inspired by Peter Landin's ISWIM language from the mid 1960s [1966b], which was, in turn, a sugared syntax for Church's lambda calculus. The ISWIM framework provided higher-order functions, which were used to express proof tactics and their composition. To ISWIM were added a static type system that could guarantee that programs that purport to produce *theorems* in the object language actually produce logically valid theorems and an exception mechanism for working with proof tactics that could fail. ML followed ISWIM in using strict evaluation and allowing impure features, such as assignment and exceptions.

Initially the ML language was only available to users of the LCF system, but its features soon evoked interest in a wider community as a potentially general-purpose programming language. Cardelli, then a graduate student at Edinburgh, decided to build a compiler for a free-standing version of ML, initially called "ML under VMS" (also "VAX ML" in contrast to "DEC10 ML," which was used to refer to the LCF version), that allowed people to start using ML outside of LCF. A bit earlier in Edinburgh, in parallel with later stages of the development of LCF, Rod Burstall, David MacQueen, and Don Sannella designed and implemented a related functional language called HOPE [1980][2] that shared ML's polymorphic types but added *datatypes*[3] to the type system.

[1] Milner's LCF group initially included postdocs Malcolm Newey and Lockwood Morris, followed in 1975 by Michael Gordon and Christopher Wadsworth, plus a number of graduate students.

[2] "HOPE," despite being all-caps, is not an acronym, but refers to the name of the building where the language was born: Hope Park Square.

[3] We use the term "datatypes," instead of the often used term "algebraic data types" because it is more concise and to avoid confusion with algebraically specified types, which are usually restricted to first-order types.

What makes Standard ML (SML) an interesting and important programming language? For one thing, it is an exemplar of a strict, statically typed functional language. In this rôle, it has had a substantial influence on the design of many modern programming languages, including other statically-typed functional languages (e.g., OCaml, F#, Haskell, and Scala). It also gave rise to a very substantial research community and literature devoted to investigation of the foundations of its type and module system. The focus of this research community broadened in the 1990s to encompass the foundations of (typed) object-oriented programming languages, as represented by the work of Luca Cardelli, John Mitchell, Kim Bruce, Benjamin Pierce, and others.

More particularly, it is worth understanding how Standard ML pioneered and integrated its most characteristic *features*.

- Static typing with type inference and polymorphic types, now commonly known as the Hindley-Milner type system.
- Datatypes with the corresponding use of pattern matching for case analysis and destructuring over values of those types.
- A module sub-language that itself is a functional language with a substantial extension of the basic type system of the core language.
- A type safe exception mechanism providing more flexible control structures.
- *Imperative* (i.e., state mutating) programming embodied in the updatable reference construct.

This complex of features evolved through the design of the language and its early formal definition in the 1980s [Harper et al. 1987a], the first published definition [Milner and Tofte 1991; Milner et al. 1990], and were further refined in the revised definition published in 1997 [Milner et al. 1997] and the Basis library standard [Gansner and Reppy 2004]. Theoretical foundations were investigated and extensions and variants were explored in multiple implementations of the language. Extensions have included first-class continuations [Harper et al. 1993b] primitives to support concurrent programming [Reppy 1991], and polymorphic record selection [Ohori 1992].

Another contribution of Standard ML was the emphasis on *safety* in programming, meaning that the behavior of all programs, including buggy programs, would be fully determined by a defined semantics of the language. Another way of characterizing safety is that programs cannot "crash" and fail in unpredictable ways. In practice, safety was achieved by a combination of static type checking and a run-time system that implements a safe memory model. Earlier languages, such as Lisp, achieved a weaker form of safety with respect to primitive data structures of the language, but the safety guarantees could not be extended to non-primitive abstractions such as binary search trees.

Standard ML has also driven advances in compiler technology. Standard ML implementations have exploited new techniques such as the use of continuation passing style and A-normal forms in intermediate languages [Appel 1992; Tarditi 1996]. Another new technology involved type-driven compilation and optimization, as represented by the TIL compiler at CMU [Tarditi et al. 1996] and the FLINT project at Yale [Shao 1997; Shao and Appel 1995] that translate type information in the source language into typed intermediate languages and in extreme cases all the way to executable code.

Finally, a distinctive attribute of Standard ML is that the language was intended to be formally defined from the outset. Most languages (including other languages in the ML family such as OCaml and F#) are defined by their implementation, but it was a major theme in the British programming research culture that this was not good enough and that one should strive for mathematical rigor in our understanding of what a programming language is and how programs behave. While there were previous examples of languages with semi-rigorous definitions (e.g., McCarthy's Lisp [1960] and Algol 68 [Lindsey 1996; Mailloux et al. 1969; van Wijngaarden et al. 1969]), these efforts were

post hoc. The formal definition of Standard ML differs from these earlier efforts in several respects; most importantly, in that it was a primary goal of the designers of the language.

The remainder of the paper is organized as follows. We start with a discussion of the developments that led up to the idea of defining a "Standard ML" in 1983, including a brief look at the history and character of British programming language research, followed by a discussion of the immediate precursors of Standard ML, namely LCF/ML, HOPE, and Cardelli ML (or "ML under VMS").

We then give an overview of the history of design process and the personalities involved in Section 3. This section explores the problems and solutions that arose during the evolution of the design from 1983 through 1986; i.e., what were the key design decisions, and how were they made?

The next four sections focus on specific aspects of the language. First, Section 4 describes the type system for the core language. Then, Section 5 focuses on the particular problems of design and definition involved in the module system, which was, in some ways, the most novel and challenging aspect of the language, and one which has produced a substantial research literature in its own right. Section 6 deals with the history of how the formal Definition of Standard ML was created from 1985 through 1989, leading to the publication of the 1990 Definition [Milner et al. 1990].[4] After the completion of the Definition, and even before, reseach continued on finding better type-theoretical foundations for the static semantics, and also toward a more feasable and ideally mechanized metatheory (proofs of safety and soundness). These developments are covered in Section 7. An important part of the revised definition of SML was the standardization of a library of essential modules for serious application development and interaction with the computing environment (operating system, networking, etc..). This effort is described in Section 8.

Section 9 concludes the paper with a discussion of several topics. These include mistakes in the design of the language, a survey of its impact on programming language research, and its application in the field. We also discuss the debate over whether the language design should be fixed by the 1997 Revised Definition or be allowed to evolve. Lastly we describe the current state of affairs.

2 BACKGROUND

In this section, we describe the technical and cultural context in which Standard ML was created. Part of this context was formed by the culture and personalities of British programming language research in the 1960s and 1970s. An important part is played by the set of precursor programming languages developed at the University of Edinburgh, which directly informed and contributed to the design of SML; namely the ML "Meta Language" of the LCF theorem proving system, the HOPE language, and Cardelli's dialect of ML.

ML was designed in a British culture of programming language theory and practice, and more specifically an Edinburgh culture, that prevailed during the 1970s and 80s. There was a large and active artificial intelligence community at Edinburgh that had a strong association with early work on programming languages. This community was familiar with Lisp and its applications in AI, but was also a center of logic programming and with an active Prolog user community. POP-2 [Burstall et al. 1977] was a home-grown AI and symbolic programming language.

The British programming language scene at the time was also strongly influenced by the work of Christopher Strachey and Peter Landin. Strachey had developed the CPL conceptual language [1963; 1966] and had collaborated with Dana Scott on the development of denotational semantics [Scott and Strachey 1971], whose metalanguage was essentially a variety of typed lambda calculus. Strachey also coined the term *parametric polymorphism* [1967] for situations where a definition or construct is parameterized relative to a type. Landin had promoted the use of the lambda calculus as a basis for

[4]Hereafter we refer to the 1990 definition as the "Definition."

programming via the ISWIM conceptual language with an operational evaluation semantics [1964; 1966b], and as the target language for a translational semantics of Algol 60 [1965a; 1965b]. Also worth noting is Tony Hoare's early work on abstract data types and language design, which was well known to the Edinburgh community. A common thread of most of the British language research at the time was an effort to formalize the basic concepts in programming languages and use these formalizations for analysis, proof of metalinguistic properties, and as a guide to "principled" language design.[5]

2.1 LCF/ML — The Original ML Embedded in the LCF Theorem Prover

We start with a brief look at some programming languages that influenced the design of LCF/ML.

Lisp[6] was the language used in the Stanford LCF project [Milner 1972] that Milner had developed in collaboration with Richard Weyhrauch and Malcolm Newey before coming to Edinburgh, and it was familiar to all the ML design participants as the then *de facto* standard language for symbolic computing. Lisp was a functional language that could manipulate functions as data, although its behavior as a functional language was flawed by the use of dynamic binding. It provided lists as a general, primitive type and had an escape mechanism (catch and throw). Besides the fact that it lacked proper function closures (true first-class functions), its main drawback as a proof assistant metalanguage was that it was dynamically typed and had no support for abstract types. Lisp was the language in which the Stanford LCF system was implemented and also served as its "meta-language." The design of ML for Edinburgh LCF was informed by Milner's experience at Stanford; in fact, LCF/ML was implemented by translation into Stanford Lisp code.

ISWIM[7] was a conceptual language created by Landin as a general framework for future programming languages [1966b] and as a tool for studying models of evaluation [1964]. ISWIM was basically an applied, call-by-value lambda calculus with syntactic sugar (e.g., "`where`" and "`let`" declaration constructs) for convenience. Landin also proposed an escaping control construct called the J operator that was originally designed to model gotos in Algol 60 [1998]. Although ISWIM was not statically typed, Landin used an informal, but systematic, way of describing data structures such as the abstract syntax of ISWIM itself. This informal style of data description was a precursor of the datatype mechanism introduced in HOPE and inherited by Standard ML.

Landin mentions a prototype implementation of ISWIM in [1966b], and he and James Morris (at MIT) implemented an extension of ISWIM as the language PAL (Pedagogical Algorithmic Language) [Evans, Jr. 1968a,b, 1970; Wozencraft and Evans, Jr. 1970].[8]

POP-2 was designed by Robin Popplestone and Burstall in the late 1960s [1977; 1968; 1968a], following on Popplestone's earlier POP-1 language [1968b]. Like Lisp, POP-2 was a functional language with dynamic binding and dynamic (i.e., runtime) typing, but its syntax was in the ALGOL-style, and its features were heavily influenced by the work of Strachey and Landin. It had a number of notable features such as a form of function closure via partial application and an escape mechanism (the *jumpout* function) inspired by Landin's J operator that plays a rôle similar to catch/throw in Lisp. The design of ML was also influenced by POP-2's record structures and by

[5]It is interesting to note that most of the central personalities first met through an unofficial reading group formed by an enthusiastic amateur named Mervin Pragnell, who recruited people he found reading about topics like logic at bookstores or libraries. The group included Strachey, Landin, Rod Burstall, and Milner, and they would read about topics like combinatory logic, category theory, and Markov algorithms at a borrowed seminar room at Birkbeck College, London. All were self-taught amateurs, although Burstall would later get a PhD in operations research at Birmingham University. Rod Burstall was introduced to the lambda calculus by Landin and would work for Strachey briefly before moving to Edinburgh in 1965. Milner had a position at Swansea University before spending time at Stanford and taking a position in Edinburgh in 1973.
[6]Originally spelled "LISP," for LISt Processing.
[7]A quasi-acronym for "If you see what I mean."
[8]PAL development was continued by Arthur Evans and John M. Wozencraft at MIT.

its *sections*, a simple mechanism for name-space management that provided rudimentary support for modularity.

GEDANKEN [Reynolds 1970] was another conceptual language defined by John Reynolds based on the principle of completeness (meaning all forms of values expressible in the language were "first-class," in the sense that they could be returned by functions and assigned to variables), and the concept of references as values that could contain other values and be the targets of assignments. GEDANKEN also included labels as first-class values, a feature that enabled unconventional control constructs.

We now proceed to ML as it existed in the interactive LCF theorem proving system, where it served as the metalanguage (i.e., proof scripting language) and command language. We refer to this original version of the language as LCF/ML. The most important feature of LCF/ML was its static type system, with (parametric) polymorphic types and automatic type inference [Milner 1978]. This type system is explored in detail in Section 4.

A major benefit of this typing algorithm was that it did not require the programmer to explicitly specify types (with rare exceptions), but instead inferred the types implicit in the code. This feature preserves some of the simplicity and fluidity of dynamically typed languages like Lisp and POP-2, and was particularly important for LCF/ML's use as a scripting and command language.

2.1.1 Control Structures. Like ISWIM, LCF/ML was at its core a call-by-value applicative language with basic expressions composed using function application. On top of this base functional language were added imperative features in the form of assignable variables, with a primitive assignment operator to update their contents, iterative control structures (loops), and failure (i.e., exception) raising and trapping constructs.

The language offered an interesting combined form for conditionals and loops, where either the **then** or **else** keywords of a standard conditional expression could be replaced by **loop** to turn the conditional into an iterative statement. These conditional/loop expressions could also be nested. As an example, here is the code for the gcd function:

```
let gcd(x,y) =
    letref x,y = x,y
    in if x > y loop x := x - y
       if x < y loop y := y - x
       else x;;
```

In this example, control returns to the top of the whole conditional after executing either of the **loop** branches. The expected **else** keyword following the first loop can be elided, as it is here. This example also illustrates the **letref** declaration form for introducing assignable variables.

2.1.2 Polymorphism and Assignment. A notable complication in the type inference algorithm for ML arose from the interaction between polymorphic types and assignable variables introduced by **letref** declarations. It was soon discovered that without some sort of restriction on polymorphic typing of such variables, the type system would be unsound. One rather *ad hoc* and unintuitive solution was devised for LCF/ML, but the search for better solutions continued for many years as described in some detail in Section 4.4.

2.1.3 Failure and Failure Trapping. Another feature of LCF/ML that was motivated by the needs of programming proof tactics was an exception or *failure* mechanism. Failures were identified by *tokens*, which were strings of characters enclosed in backquotes. To signal failure, one used the syntax "**failwith** t," which would signal a failure/exception labeled by the token t, and failures could be trapped using the syntax "e_1 ? e_2," where a failure in e_1 would result in the expression e_2 being

evaluated. One could also specify a list of failure tokens to trap using the syntax "e_1 ?? *tokens* e_2," where *tokens* was a space-separated sequence of strings enclosed in double backquotes. A novel aspect of the failure mechanism was the iterative form of failure trapping (using "!" and "!!" instead of "?" and "??") These forms would repeat execution of the expression e_1 after the handler e_2 had run, until e_1 evaluated without failure. Proof tactics can fail for various reasons, and failure trapping is useful for defining higher-order tactics (tacticals) that can detect failures and try an alternate tactic (or the same tactic after some adjustment).

2.1.4 Types and Type Operators. The basic types of LCF/ML are `int`, `bool`, `token` (i.e., string), and a trivial type denoted by "." that has a single element, the null-tuple "()." There are also built-in types `term`, `form`, `type`, and `thm` representing the terms, formulae, types, and theorems of the PPLAMBDA logic (the *object language* of the LCF system). The built-in type constructors include binary products "ty_1 # ty_2," binary disjoint sums "ty_1 + ty_2," function types "ty_1 -> ty_2," and list types "ty `list`."

The `list` type constructor could be considered a predefined abstract type except that special syntax was provided for list expressions.

Another aspect of the type system of LCF/ML was the ability to define new types. These could be either simple abbreviations for type expressions composed of existing types and type operators, or they could be new abstract types, defined using the **`abstype`** declaration keyword. Abstract type definitions specified a hidden concrete representation type and a set of exported values and functions that provided an interface for creating and manipulating abstract values. Abstract types could be parameterized and they could be recursive (i.e., an abstract type could appear in its own concrete representation). Abstract types played an important rôle in the LCF theorem proving system, where `thm` was an abstract type and the functions it provided for building elements of the type corresponded to valid inference rules in the logic of LCF, thus ensuring by construction that a value of type `thm` must be proveable. As noted by Gordon et al. [1978b], the **`abstype`** construct is similar to the notion of "clusters" in the CLU language [Liskov et al. 1981].

2.2 HOPE

The HOPE programming language [Burstall et al. 1980] was developed by Rod Burstall's research group in the School of Artificial Intelligence at Edinburgh just after LCF/ML from 1977 to 1980. This language was inspired by the simple first-order equational language that was the subject of Burstall and Darlington's work on program transformation [1977]. Burstall had designed a toy language called NPL [1977] that added a simple form of datatype [1969] and pattern matching to the earlier equational language. David MacQueen and Burstall (later joined by Don Sannella), added LCF/ML's polymorphic types and first-class functions (lambda abstractions) to create the HOPE language. From the perspective of the later evolution of ML, the notable features of HOPE were its datatypes, pattern matching over datatypes, clausal (or equational) function definitions, and the combination of polymorphic type inference with overloading of identifiers.

2.3 Cardelli ML

We start by giving a brief history of Cardelli's dialect of ML, also known as "ML under VMS" (the title of the manual), before considering the language innovations it introduced [Cardelli 1982d].[9]

Cardelli began working on his ML compiler sometime in 1980. The compiler was developed on the Edinburgh Department of Computer Science VAX/VMS system, so Cardelli sometimes informally called his version "VAX ML" to distinguish it from LCF/ML, which was also known

[9]Note that Cardelli was building his VAX ML at the same time as he was doing research for his PhD thesis [Cardelli 1982a]. He also developed his own text formatting software, inspired by Scribe, that he used to produce his thesis and the compiler documentation, and a simulation of the solar system!

as "DEC-10 ML," because it ran under Stanford Lisp on the DEC-10/TOPS-10 mainframe [Cardelli 1981].[10]

Cardelli's preferred working language at the time was Pascal, so both the compiler and the runtime system were written in Pascal, using Pascal's unsafe union type to do coercions for low-level programming (e.g., for the garbage collector).[11] Cardelli's ML compiler for VAX/VMS generated VAX machine code, and was much faster than the LCF/ML (DEC-10) version, which translated ML to Lisp code, which was interpreted by the host Lisp system. The compiler was working and was made available to users in the summer of 1981 (version 12-6-81, distributed on June 12, 1981), although a working garbage collector was not added until version 13-10-81.

The earliest surviving documents relating to the compiler date to late 1980: "The ML Abstract Machine," a description of the abstract machine AM [Cardelli 1980a] (which would develop into the FAM [Cardelli 1983a]), and "A Module Exchange Format," a description of an external string format for exporting ML runtime data structures [Cardelli 1980b]. There is a README file titled "Edinburgh ML" from March 1982 that describes how to install and run the system [Cardelli 1982b], and a partial manual titled "ML under VMS" providing a tutorial introduction to the language [Cardelli 1982d], corresponding to Section 2.1 of "Edinburgh ML" [Gordon et al. 1979].

In early 1982, Nobuo Saito, then a postdoc at CMU, ported Cardelli's ML compiler to Unix, using Berkeley Pascal [1982]. In April 1982, Cardelli completed his PhD at Edinburgh [1982a] and moved to the Computing Science Research Center (the birthplace of Unix) at Bell Labs, and immediately began his own Unix port, which was available for distribution in August 1982. The runtime system for the Unix port was rewritten in C, but most of the compiler itself remained in Pascal. The first edition of the *Polymorphism* newsletter[12] contained a list of known distribution sites [Cardelli 1982c]. At that time, there were at least 23 sites spread around the world, several using the new Unix port. The Unix port had three releases during 1982 (13-8-82, 24-8-82, and 5-11-82), accompanied with some shifts in language design and system features, notably a new type checker for ref types and an early version of file I/O primitives. The manuals for these releases were all titled "ML under Unix," instead of "ML under VMS."

The next major milestone was the first Standard ML meeting in Edinburgh in April 1983 (see Section 3.1). Cardelli agreed to a request from Milner to suspend work on the manual for his ML dialect (now titled "ML under Unix") pending developments in response to Milner's initial proposal for Standard ML [Milner 1983d]. Following the meeting Cardelli began to change his compiler to include new features of the emerging Standard ML design, resulting in Pose 2 (August 1983),[13] Pose 3 (November 1983), and finally Pose 4 (April 1984). This last version is described in the paper "Compiling a Functional Language" [Cardelli 1984a].

The first description of the Cardelli's version of ML was a file mlchanges.doc [Cardelli 1981] that was part of his system distribution. This file describes the language by listing the changes made relative to LCF/ML (DEC-10 ML). The changes include a number of minor notational shifts. For instance, LCF/ML used "`.`" (period) for list cons, while Cardelli ML initially used "`_`," later shifting to the notation "`::`" used in POP-2 [Burstall and Popplestone 1968]. The trivial type (called "`unit`" in Standard ML) was denoted by "`.`" in LCF/ML and by "`triv`" in Cardelli ML. A number of features

[10] Cardelli sometimes referred to these implementations as two varieties of "Edinburgh ML," but that name was used later for the self-bootstrapping port of Cardelli ML implemented by Kevin Mitchell and Alan Mycroft.

[11] Since the entire system was written in Pascal, there was no sharp distinction between the compiler and the runtime, which was simply the part of the system responsible for executing the abstract machine instructions (FAM code).

[12] *Polymorphism – The ML/LCF/Hope Newsletter* was self published by Cardelli and MacQueen at Bell Laboratories and distributed by physical mail to interested parties. Electronic copies are available at either http://lucacardelli.name/indexPapers.html or http://sml-family.org/polymorphism.

[13] Cardelli called his compiler versions "Poses" adopting a terminology from dance.

of LCF/ML were omitted from Cardelli ML, e.g., the "`do`" operator, sections, and the "`!`" and "`!!`" looping failure traps [Gordon et al. 1979, Chapter 2].

But the really interesting changes in Cardelli ML involved new labelled record and union types, the ref type for mutable values, declaration combinators for building compound declarations, and modules. We describe these in the following subsections.

2.3.1 Labelled Records and Unions. Cardelli was of course familiar with the conventional record construct provided in languages such as Pascal [Jensen and Wirth 1978, Chapter 7]. But inspired by Gordon Plotkin's lectures on domain theory, Cardelli looked for a purer and more abstract notion, where records and discriminated union types were an expression of pure structure, representing themselves without the need of being declared and named. The notation for records used *decorated parentheses*:

$$(|\, a_1 = e_1 ; \ \ldots \ ; \ a_n = e_n \,|) \ : \ (|\, a_1 : \ t_1 ; \ \ldots \ ; \ a_n : \ t_n \,|)$$

where e_i is an expression of type t_i. The order of the labelled fields in a record type did not matter — any permutation represented the same type.

Accessing the value of a field of a record was done using the conventional dot notation: `r.a`, where `r` is a record and `a` is a label. Records could also be deconstructed in declarations and function arguments by pattern-matching with a record *varstruct* (pattern), as in the declaration:

```
let (|a=x; b=y|) = r
```

From the beginning, Cardelli included an abbreviation feature for record patterns where a field name could double as a default field variable, so

```
let (|a; b|) = r
```

would implicitly introduce and bind variables named `a` and `b` to the respective field values in `r`. All these features of the VAX ML record construct eventually carried over to Standard ML, but with a change in the bracket notation to use `{...}`.

Labelled unions were expressed as follows, using decorated square brackets:

$$[|\, a_1 = e_1 \,|] \ : \ [|\, a_1 : \ t_1 ; \ \ldots \ ; \ a_n : \ t_n \,|]$$

The union type to which a given variant expression belonged had to be determined by the context or given explicitly by a type ascription. Variant varstructs could be used in varstructs for declaration and argument bindings, with their own defaulting abbreviation where a `[|a|]` stood for `[|a = ()|]`, both in varstructs and expressions, which supported an enumeration type style.[14] A case expression based on matching variant varstructs was used to discriminate on and deconstruct variant values, with the syntax

```
case e
  of [| a1 = v1 . e1;
        ...
        an = vn . en
     |]
```

where e is of type $[|\, a_1 : t_1 ; \ldots ; \ a_n : t_n |]$ and the v_i have type t_i in e_i.

[14] Initially, the record and variant abbreviation conventions were also applied to types, but this was not found useful and was quickly dropped.

2.3.2 The `ref` *Type.* In LCF/ML, mutable variables could be declared using the **`letref`** declaration. Cardelli replaced this declaration form with the `ref` type operator with its interface consisting of the operations `ref`, `:=`, and `!`. This approach was carried over unchanged into Standard ML, though the issue of how `ref` behaved relative to polymorphism took many years to resolve, as is discussed below in Section 4.4.

2.3.3 Declaration Combinators. Another innovation in Cardelli ML was a set of (more or less) independent and orthogonal declaration combinators for building compound declarations [Cardelli 1982d, Section 1.2]. These combinatorsare

- **`enc`**, for sequential composition, equivalent to nesting **`let`**-bindings: "`d1` **`enc`** `d2`" yields the bindings of `d1` augmented with or overridden by the bindings of `d2`.
- **`and`**, for simultaneous or parallel composition, usually used with recursion.
- **`ins`**, for localized declarations: "`d1` **`ins`** `d2`" yields the bindings of `d2`, which are evaluated in an environment containing the bindings of `d1`.
- **`with`**, a kind of hybrid providing the effect of **`enc`** for type bindings and **`ins`** for value bindings; usually used with the special "`<=>`" type declaration to implement abstract types.
- **`rec`**, for recursion

There were also reverse forms of **`enc`** and **`ins`** called **`ext`** and **`own`** for use in **`where`** expressions, thus "**`let`** `d1` **`enc`** `d2` **`in`** `e`" is equivalent to "`e` **`where`** `d2` **`ext`** `d1`."

This "algebra" of declarations (possibly inspired by ideas in Robert Milne's PhD thesis [1974]) was interesting, but in programming, the combinators would normally be used in a few limited patterns that did not take advantage of the generality of the idea. Indeed, certain combinations seemed redundant or problematic, such as "**`rec`** **`rec`** `d`," or "**`rec`** `d1` **`ins`** `d2`" (**`rec`** syntactically binds weaker than the infix combinators).

Cardelli factored the **`abstype`** and **`absrectype`** of LCF/ML using **`with`** (and possibly **`rec`**) in combination with a special type declaration form "`tname` `<=>` `texp`" that produced an opaque type binding[15] of `tname` to the type expression `texp` together with value bindings of two isomorphism functions:

```
abstname : texp -> tname
reptname : tname -> texp
```

A compound declaration "`tname` `<=>` `texp` **`with`** `decl`" would compose the type binding of `tname` with `decl` while localizing the bindings of `abstname` and `reptname` to `decl`. Thus **`with`** acted like **`enc`** at the type level and **`ins`** at the value level. This mechanism was, in principle, more general than **`abstype`**/**`absrectype`** in LCF/ML, in that the declaration `d1` in `d1` **`with`** `d2` was arbitrary and not restricted to a possibly recursive simultaneous set of isomorphism (`<=>`) type bindings, but it was not clear how to exploit increased generality.

In the end, the **`and`** combinator was used in Standard ML, but at the level of value and type bindings, not declarations (just as it was used in LCF/ML), the **`ins`** combinator became the "**`local`** `d` **`in`** `e` **`end`**" declaration form, the **`rec`** combinator was adopted, but at the level of bindings,[16] and the `<=>`, **`with`** combination were replaced by a variant of the old **`abstype`** declaration, but using the datatype form for the type part and restricting the scope of the associated data constructors.

In later versions, starting with "ML under Unix," Pose 2 [Cardelli 1983b], another declaration form using the **`export`** keyword was added. A declaration of the form

`export` *exportlist* **`from`** *decl* **`end`**

[15] Meaning that `tname` was not equivalent to `texp`.
[16] Arguably, the syntax was still too general, since it allowed an arbitrary number of **`rec`** annotations to be applied to a binding.

produced the bindings of *decl*, but restricted to the type and value names listed in the *exportlist*. Exported type names could be specified as abstract (in ML under Unix, Pose 4) meaning that constructors associated with the type were not exported. Thus both `local` and `abstype` declarations could be translated into export declarations.

2.3.4 Modules. LCF/ML had no module system; the closest approximation was a `section` directive that could delimit scope in the interactive top-level. Since Cardelli ML aimed to support general purpose programming, Cardelli provided a basic module system. A module declaration was a named collection of declarations. Modules were independent of the environment in the interactive system, and explicit import declarations were required to access the contents of other modules (other than the standard library of primitive types and operations, which was pervasively accessible, i.e., implicitly imported into every module). Cardelli called this feature a *module hierarchy*. Compiling a module definition produced an external file that could be loaded into the interactive system or accessed by other modules using the import declaration. Importing (loading) a module multiple times would only create one copy, so two modules `B` and `C` that both imported a module `A` would share a single copy of `A`. The export declaration was commonly used to restrict the interface provided by a module. There was no way to separately specify interfaces (thus no equivalent of *signatures* in Standard ML).

3 AN OVERVIEW OF THE HISTORY OF STANDARD ML

In this section, we survey the history of the design and evolution of Standard ML (which we will free to abbreviate as SML from here on). This section provides a historical framework for the more technically focused sections that follow.

3.1 Early Meetings

3.1.1 November 1982. By the fall of 1982, interest in the functional-programming systems developed in Edinburgh (LCF/ML, Cardelli ML, and HOPE) had spread through the British programming research community and beyond (to INRIA in France in particular). This interest led to the convening of a meeting titled "ML, LCF, and HOPE" at the Rutherford Appleton Laboratory (RAL) in mid-November of 1982 under the sponsorship of the SERC[17] Software Technology Initiative, chaired by R. W. Witty [Wadsworth 1983]. The 20 attendees included a large contingent from Edinburgh (Robin Milner, Rod Burstall, etc.) and representatives from RAL, Oxford, Imperial College, and Manchester.

The discussions at this meeting focused on strategy for further development of ML, LCF, and HOPE, including new implementations and ports of existing implementations. Because there were already multiple implementations (forks) of both ML (the LCF/ML embedded metalanguage and Cardelli ML), LCF (new implementations with variations at Cambridge/INRIA and Chalmers), and HOPE (the original POP-2 implementation, MacQueen's Lisp port, and work on compilers for the Alice machine in Darlington's group at Imperial College), there was concern about duplication and dispersion of effort among different design and implementation forks.

The question of merging the ML and HOPE languages to reduce duplication came up, and, interestingly, Milner's response was reported by Chistopher Wadsworth as follows:

> There was some discussion about the merits of propagating two languages (HOPE and ML). R. Milner summarized the general view that HOPE and ML represent two points in a spectrum of possible (typed) functional languages. Differing aims had led to different compromises in the design of the two languages and both

[17] Science and Engineering Research Council, the British equivalent of the National Science Foundation in the US.

> should continue to be developed. The ultimate functional language could not be envisaged yet — variety and experimentation, rather than standardization, were needed at this stage.

Thus it is clear that Milner was not contemplating a merged language at this point. Bernard Sufrin (who attended the RAL meeting), reports, however, that he had further discussions with Milner at another meeting (possibly in York) that fall where he urged Milner to consider a new, unified version of ML that might subsume Cardelli's dialect and even HOPE. Milner refers to Sufrin's suggestion in the second draft proposal [1983c].

The notes for the November 1982 meeting, along with a response from Michael Gordon and Larry Paulson (Cambridge University), who were not able to attend the meeting, were published in *Polymorphism* [Wadsworth 1983].

3.1.2 April 1983. By April 1983, Milner had written a manuscript proposal for a new language that he called "Standard ML" [1983d].[18] Milner emphasised that he was thinking of a conservative revision of the original ML design that was not intended to introduce novel features, but rather to *consolidate* ideas that had developed around ML. The language was presented as a minimal "Bare ML" that was then fleshed out with standard abbreviations and alternate forms that could be defined in terms of Bare ML. In the Bare ML language, notable changes to LCF/ML included the introduction of HOPE-style algebraic datatypes, pattern matching over datatype constructors, and clausal (or equational) function definitions.

Cardelli's labeled records were not adopted, leaving the binary product types and pairing operator of LCF/ML in place. Similarly, Cardelli's labeled variants were regarded as redundant given that datatypes provided tagged unions, with the datatype constructors serving as tags, and datatypes could also be used to define the equivalent of LCF/ML's binary tagged union operator on types.

The elaborate conditional/iterative control constructs of LCF/ML were dropped and case expressions and simple conditionals were derived using application of a clausal function. Similarly, the elaborate conditional/iterative failure trapping construct was replaced by simple `escape` and `trap` expression forms, with `escape` transmitting a *token* (that is, a string) as before.

Serendipitously, it happened that in the Spring of 1983 there was a significant group of PL researchers gathered in Edinburgh,[19] who met in the parlor of Milner's home for three days in April to discuss Milner's proposal.

3.1.3 June 1983. Following the meetings at Milner's home, there was a period of further discussion, mostly via the post,[20] and the circle of discussion widened somewhat to include Gérard Huet at INRIA and Stefan Sokolowski at the Polish Academy of Science in Gdansk.

The discussions at the meeting and after resulted in Milner's producing and distributing a longer, revised version of his proposal in June [1983a; 1983c], which was followed by further discussion. This second proposal added equality types and equality polymorphism to the language, using a restricted form of bound type variables.

Further discussion lead to the first typewritten design proposal in November of 1983; the first two drafts had been handwritten. The introduction of this draft notes that the issues of input/output

[18] The name "Standard ML" was not viewed as permanent, and it did not imply a formal *standardization* process, but simply an attempt to re-unify the dialects that had come to exist, namely LCF/ML and Cardelli ML. The fact that the name stuck was unfortunate, since in the end a new language was produced, and there was no sanctioned "standard."

[19] Participating: Rod Burstall, Luca Cardelli (Bell Labs), Guy Cousineau (INRIA), Michael Gordon (Cambridge), David MacQueen (Bell Labs), Robin Milner, Kevin Mitchell, Brian Monahan, Alan Mycroft, Larry Paulson (Cambridge), David Rydeheard, Don Sannella, David Schmidt, John Scott; unless otherwise noted, all from Edinburgh.

[20] None of the institutions involved were connected to the Arpanet, and email was not yet a routine means of communication.

and separate compilation were contentious and, thus, the proposal focuses on defining a "core" language.

3.1.4 June 1984. In June 1984, Milner organized a slightly more formal second meeting in Edinburgh to refine and extended the design. This second meeting was more planned than the first one, with Milner distributing an invitation and timetable [1984a] in advance of the meeting. A meeting report was written by MacQueen and Milner, and published in the *Polymorphism* newsletter the following January [1985]. The meeting also resulted in the publication of two papers — one on the Core language [Milner 1984b] and a second on the module system [MacQueen 1984] — at the 1984 Lisp and Functional Programming Conference in Austin, Texas.

3.1.5 May 1985. The third design meeting was a three-day affair held in Edinburgh in May 1985. This meeting was more elaborate, with a number of position papers being presented [Milner 1985b]. A meeting report was prepared by Robert Harper [1985].

A major topic of discussion at the third meeting, which consumed the whole first day, was a proposal by MacQueen for a Standard ML module system. This initial proposal was based on MacQueen's earlier conceptual work on a module system for HOPE [1981]. The design was inspired partly by work on parametric algebraic specifications, particularly Burstall and Goguen's Clear specification language [1977], and partly by research in type theory and type-based logics (e.g., by Per Martin Löf [1982; 1984]). The main elements of the design were signatures (interface specifications), modules (implementations of interfaces), and parametric modules, called *functors*. Modules could be nested inside other modules and functors.

The design issues involved in the module system proved to be surprisingly subtle, and much discussion went on during the period from 1984 through 1988 on the details of the design and the methods for its formal definition. Major issues were coherence (sharing) among functor parameters, the nature of signature matching (whether it was coercive or not), and the static identity of modules (i.e., are module declarations generative). These issues are discussed in detail in Section 5.

While modules were the dominant topic of discussion at this meeting, an afternoon was given over to discussing other topics. These included early proposals for some form of labeled tuples (what would eventually become the *record* type of SML), and various syntax issues.

The 1985 meeting was also the first meeting that did not include participation from the FORMEL group at INRIA. By this time, they had decided to develop a separate dialect of ML that would be free from the need to adhere to a standard. This version of ML became the Caml (and eventually OCaml) branch of the language; the first release was in 1987 and compiled down to the virtual machine of the LeLisp system (also developed at INRIA) [OCaml-history 2019].

3.2 The Formal Definition

Up to this point, the design proposals and discussion were carried out on an informal basis, with considerations of formal semantics in the background. But with the arrival of Robert Harper in Edinburgh in the Summer of 1985, work began on the semantics of the language and subsequent design work was generally integrated with that effort.

The development of the formal definition of Standard ML began after the May 1985 meeting. It was primarily developed by Milner, Mads Tofte, and Harper, with input from Dave MacQueen and other participants in the design process, including Burstall, Kevin Mitchell, Alan Mycroft, Lawrence Paulson, Don Sannella, John Scott, and Christopher Wadsworth. By April 1985, Milner had produced a third draft of "The Dynamic Operational Semantics of Standard ML" [1985a]. This was an evaluation semantics ("big step") in the same operational style eventually used in the Definition. Don Sannella had also developed a denotational semantics of the module system [Sannella 1985], which revealed some issues and ambiguities in the informal description [Milner et al.

1990, Appendix E]. Harper wrote the first version of a static semantics in the summer of 1985, but the first published version of the static semantics appeared in 1987, including the characterization of principal signatures for modules [Harper et al. 1987b]. The development of the semantics was undertaken as Tofte's Ph.D. dissertation research [1988], which examined many issues in the language itself and in the semantic methodology needed to define it. Milner and Tofte extensively developed these ideas into *The Definition of Standard ML*, which was published in 1990 [Milner et al. 1990]. A companion volume, *Commentary on Standard ML* [Milner and Tofte 1991], was published shortly thereafter by Milner and Tofte. The Commentary provided an analysis of design alternatives and fundamental properties of the definition, particularly the static semantics. These two volumes served as the foundation for some subsequent implementations, most notably the *ML Kit Compiler* [Birkedal et al. 1993], and spurred a large body of research on the language's metatheory and on the methodology of formal definition. These topics are explored further in Section 6 and Section 7.

3.3 Standard ML Evolution

By the early 1990s, a great deal of work had gone into the design, formal definition, and implementation of Standard ML, and substantial applications had been built (including Standard ML compilers written in Standard ML). Some technical bugs, gaps, and ambiguities had been discovered in the Definition, and some problems or weaknesses in the design itself had become evident. Thus people in the community were beginning to feel that at least a minor revision was warranted to correct acknowledged mistakes.

3.3.1 ML 2000. In January 1992, at the POPL conference in Albuquerque, New Mexico, a small group consisting of Robert Harper, John Mitchell, Dave MacQueen, Luca Cardelli, and John Reppy[21] had lunch together and began to discuss whether the time was right to think about a next generation of ML, based on the accumulated experience with design, implementation, use, and theoretical developments related to ML (both SML and Caml).

The idea was more ambitious than to just make minor corrections or modifications to Standard ML. The group was open to a more radical redesign that might even re-integrate the Caml fork, by trying for a language design that would be seen as an improvement over both Standard ML and Caml.

This initial meeting lead to a series of meetings (roughly two per year, usually connected with either POPL, ICFP, or IFIP Working Group 2.8, but with a couple *ad hoc* meetings at a cabin at Lake Tahoe) with a gradually expanding group of interested researchers, including Xavier Leroy and Didier Rémy from the Caml community.

A number of relatively minor technical issues were discussed, but it turned out that the central question was whether ML would be improved by adding object-oriented features. At this time, Cardelli and Mitchell were both involved in developing type-theoretic foundations for object-oriented languages, and Didier Rémy and his student Jérôme Vouillon were beginning work that would result in Objective Caml (first released in 1996) [OCaml-history 2019]. Kathleen Fisher and Reppy would also explore the idea extending ML with object-oriented features in the Moby language [1999; 2002]. But another point of view, espoused by Harper and MacQueen among others, was that the ideas underlying object-oriented programming were both excessively complex and also unsound from a methodological point of view [MacQueen 2002].

The dispute between the advocates and the opponents of object-oriented features was never resolved, leading to a failure of the group to produce a new language design. Jon Riecke, however,

[21] The exact cast of characters is uncertain — memories differ.

was able to write up a report on the ideas on which there was general agreement [ML2000 Working Group 1999]. In this report, ML2000 was envisioned as having the following properties:

- A module system based on SML modules, but with a well-defined semantics of separate compilation and support for higher-order functors. There were, however, competing views on how to realize the latter mechanism, with some arguing for the approach taken in SML/NJ [MacQueen and Tofte 1994], whereas others favored the "applicative functor" approach taken in OCaml [Leroy 1995].
- Support for object-oriented programming, including a notion of object types and a subtyping relation. Type specifications in a signature could specify subtyping relationships (i.e., that an abstract exported type was a subtype of some other type).
- Concurrency would be included using a design based on the `event` type constructor and supporting combinators found in Concurrent ML [Reppy 1999]. There was also an awareness that the semantics of ML2000 would have to take into account relaxed memory-consistency models; an idea that has become standard in recent years.
- Instead of the single extensible datatype (`exn`) in SML, there would be a general mechanism for defining "open" datatypes based on a design by Reppy and Riecke [1996].
- In addition, there were various minor changes, such as adopting modulo arithmetic and eliminating polymorphic (or generic) equality.

3.3.2 The Revised Definition (Standard ML '97). Within the core SML community, the ML 2000 discussions did bear concrete fruit.

By the mid 1990s, significant experience with the Standard ML design had been accumulated through the work on the formal definition, multiple implementation efforts, and the development of reasonably large software systems written in SML. A core group of designers, Milner, Tofte, Harper, and MacQueen,[22] felt it would be valuable to attempt a modest revision of the language design, leading to the creation of Standard ML '97 and the publication of the *Definition of Standard ML (Revised)*. This revision simplified the Core type system in significant ways, notably by introducing the *value restriction rule* to control the interaction of polymorphism and mutable data, eliminating the need for "imperative" type variables that were used to express a weakened form of polymorphism [Wright 1995].

It also both simplified and enriched the module system. Signatures were made more expressive by introducing definitional (or translucent) type specifications, which had been implemented in Standard ML of New Jersey in 1993 (Version 0.93) and proposed in papers by Harper and Lillibridge [1994] and Leroy [1994] at POPL 1994. Opaque signature matching (using the delimiter "`:>`") was added, and the notion of structure sharing was weakened to implied type sharing. These issues will be discussed more fully below in Section 5.

Along with changes to the language, the Basis Library was substantially revised and expanded to match common practice. New built-in types were added for characters, unsigned integers, and multiple precisions of numeric types were supported. These changes required modifications to the syntax of the language, as well as extending the overloading mechanism to numeric literals.

The discussion of these revisions took place over a period of about a year (1995-1996) and Tofte carried out the detailed revision of the semantic rules. The revised Definition [Milner et al. 1997] was published in 1997 — hence the revised language is commonly known as SML '97.

[22] Brought together in Cambridge in the fall of 1995 by the Semantics of Computation program at the Newton Institute.

4 THE SML TYPE SYSTEM

This section covers the background, design, theory, and implementation(s) of the ML type system. We start with a look at the early history of the concept of types as it arose in research in the foundations of mathematics. We then consider the issue of implicit typing and type inference, and the related notion of parametric polymorphism.

The polymorphic type inference system used in ML (and related languages) is owed to Robin Milner, who independently rediscovered the ideas that had been previously explored by logicians (Haskell Curry, Max Newman, and Roger Hindley). But Robin Milner added the essential idea of *let-polymorphism*, or introducing polymorphism at local declarations, along with his now-classic "Algorithm W" for performing type inference [1978].

The two theoretical properties that the type system strives to achieve are *soundness* and *principality*. Soundness roughly means that a statically well-typed program will not fail with a certain class of errors called type errors. Principality roughly means that the type inferred for a program or expression is as general as possible – any other valid typing would be a specialization of this most general type. We also discuss the interaction of polymorphism and effects. Handling effects in a sound fashion disrupts the theoretical simplicity of the type system and conflicts with principality. Pragmatically the problem of effects was initially addressed by complications in the type system such as *imperative type variables*. Later, in the *Definition (Revised)*, the decision was made to simplify the type system by introducing the *value restriction*, which is described below.

We then move on to consider the structure of types (products, sums, etc.). Here the major innovation in ML is the notion of *datatypes*. These were implicit in Peter Landin's informal conventions for defining data types in ISWIM [1966a; 1966b] and were developed further by Rod Burstall in his language NPL [1977], and finally in a fully developed form in HOPE [Burstall et al. 1980].

Finally we address some implementation issues and techniques related to the type system.

4.1 Early History of Type Theory

The development of the theory of types that eventually led to the Standard ML type system goes back more than a hundred years. It is covered in considerable detail and depth in references such as "A Modern Perspective on Type Theory, From its Origins until Today" [Kamareddine et al. 2004], "Lambda-Calculus and Combinators in the 20th Century" [Cardone, Jr. and Hindley 2009], and "The Logic of Church and Curry" [Seldin 2009].[23] The following is a summary of some of the most relevant ideas from this history.

In the late 19th century, Georg Cantor, Gottlob Frege, Giuseppe Peano, and others endeavored to provide rigorous logical foundations for mathematics, particularly arithmetic, but also real analysis (with Richard Dedekind and Augustin-Louis Cauchy). Cantor created his set theory [1874] in pursuit of a proof of the existence of transcendental (irrational) real numbers. Frege devised a logical calculus, called *Begriffsschrift* [1879], and used it to construct a formal foundation for arithmetic in "Grundgesetze der Arithmetik" [1893]. And, of course, Peano developed his well-known axioms for arithmetic [1889] and invented something like the modern notation used in logic.

In 1901, Bertrand Russell discovered a paradox in Frege's logic [1967]. This paradox (and others) threatened the program to build logical foundations for mathematics. Russell, with the collaboration of Alfred North Whitehead, launched an effort to repair the fault that led to the publication of the three volumes of *Principia Mathematica* between 1910 and 1913 [1910,1912,1913]. Their approach to avoiding the paradoxes was to introduce a *ramified* theory of types into a formalism derived from Frege's Begriffsschrift. This formalism was based, like Frege's, on the idea of *propositional*

[23] For general introductions to typed λ-calculus, see texts by Hindley [1997] or Henk Barendregt [1992].

functions rather than sets. The types characterized the arguments of such propositional functions: their number and "levels" (what we would today call their "order"). The ramification took the form of *orders* (natural numbers) that specified the universes quantified over in the definition of a propositional function: order 0 if there was no (universal) quantification, order 1 if there were quantifiers over "individuals" (basic values like numbers), 2 for quantifiers over order 1 propositional functions, and so on. These notions of types and orders were treated somewhat informally, and in fact there was no definition of or notation for types, nor for what it meant for a term to have a type. There was only an incomplete definition of when two terms "have the same type."[24]

This ramified theory of types was both complex and not fully defined in *Principia Mathematica*. During the 1920s, F.P. Ramsey [1926], and David Hilbert and Wilhelm Ackermann [1928] showed that the complexity of ramified types was not actually necessary to avoid the paradoxes and that a *simple type theory* would suffice [Carnap 1929]. Their versions of a simple theory of types was defined in terms of propositional functions, where the types characterized only the arguments of functions, since all such functions returned propositions (or, roughly, Boolean values). In 1940, Alonzo Church published his paper "A formulation of the Simple Theory of Types" [1940]. This paper still focused on using types (in conjunction with an applied λ-calculus enriched with logical operators like "NOT," "AND," and universal quantification) to express a logic, but it supported a more general notion of functions that could return values other than propositions (or truth values). Church's type theory became the standard in the following decades.[25]

Meanwhile, in the 1930s Haskell Curry was working on Combinatory Logic, another "logical" calculus [Schönfinkel 1924] [Curry 1930, 1932]. Curry introduced what he called *functionality* in Combinatory Logic, which consisted essentially of types for combinators and combinations [1934]. Although his notation was archaic and types were not notationally distinguished from general combinators, his notion of functionality for combinators amounts to what we now call parametric polymorphic types. For instance, his axiom for the typing of the K combinator (corresponding to the λ-expression $\lambda x.\lambda y.x$) is

$$(x, y)Fy(Fxy)K \qquad \text{(FK)}$$

where the notation $F\alpha\beta$ translates into $\alpha \to \beta$ (assuming α and β are terms representing types), and $F\alpha\beta x$ represents the typing assertion $x : \alpha \to \beta$. So the above axiom for typing K translates into the modern form

$$K : \forall(x, y)\,.\, y \to (x \to y) \qquad \text{(FK)}$$

Another important point about Curry's type theory is evident here: a function (e.g., K) can simultaneously have many different types. The *meaning* of a typing assertion such as $Fxyz$ (i.e., $z : x \to y$) is given by the following axiom:

$$(x, y, z)(Fxyz \;\Rightarrow\; (u)(xu \;\Rightarrow\; y(zu))) \qquad \text{(Axiom F)}$$

or in modern notation:

$$\forall(x, y, z)(z : x \to y \;\Rightarrow\; (\forall u)(u : x \;\Rightarrow\; zu : y)) \qquad \text{(Axiom F)}$$

in other words, $z : x \to y$ means that z maps members of type x to members of type y. Thus having a type is a property of a term that does not exclude that term having other types; i.e.; the type is not intrinsic to the term. This approach is called *Curry-style* typing in contrast with the types in

[24] Modern reformulations of the paradoxes and the theory of ramified types are given by Thierry Coquand [2018] and Fairouz Kamareddine [2004].

[25] But Church's notation for types did not survive. For instance, in his notation, the type of functions with domain α and range β was $(\beta\alpha)$, which we now would express as $\alpha \to \beta$.

Church's simply typed λ-calculus, where the type of a term is uniquely determined by the syntax of the term and the type is an intrinsic property of the term.[26]

Church's simply typed λ-calculus introduced the problem of determining whether a term in the calculus was well-formed, which is essentially the problem of *type checking*. Curry's formulation entailed the more challenging problem of discovering a *type assignment* for a given untyped term. The first solution of this type assignment problem (i.e., an algorithm for discovering a type assignment for a term) for the λ-calculus was provided by Max Newman[27] in 1943 [Hindley 2008; Newman 1943].[28]

Curry returned to the problem of discovering (or *inferring*) type assignments more than 30 years later [1969]. His approach was to use equations to express the constraints that unknown types had to satisfy. For instance, given an application $F\,A$, the types t_F and t_A of F and A have to satisfy the equation

$$t_F = t_A \rightarrow t$$

for some type t. Other equations are generated for occurrences of primitive combinators whose types are presented as type schemes for the basic combinators like K and S, which, in modern notation, are:

$$K : u \rightarrow v \rightarrow u \qquad S : (u \rightarrow (v \rightarrow w)) \rightarrow ((u \rightarrow v) \rightarrow (u \rightarrow w))$$

where the type variables in the schemes are implicitly universally quantified. So the assignment problem for a compound term generates a set of first-order order equations involving a set of type variables representing the types to be determined. Curry presented an algorithm to solve such equations that is equivalent to Robinson's Unification algorithm [Robinson 1965].[29] Curry claimed that his algorithm produced a *principal functional character* (or what we now would call a *principal type scheme*), meaning that any correct type assignment satisfying the constraints can be obtained by applying a substitution to the scheme produced by the algorithm. Curry did not prove that the algorithm produces the most general (i.e., principal) typing, however [1969]. Nevertheless, since the algorithm is equivalent to Robinson's, it does indeed have this principality property.

At about the same time, Hindley [1969] discovered the same type assignment algorithm, but in his case he explicitly invoked Robinson's unification algorithm to solve the typing equations, so his proof of principality followed from Robinson's result.

Note that in these three solutions of the type assignment problem (Newman 1943, Curry 1966/1969, and Hindley 1969), the assignment problem is posed for a complete, closed, term, with no context required. There is a good exposition of this Curry-style type assignment problem, in its purest form, in Hindley's *Basic Simple Type Theory* [1997], but it is worth explaining the intuitive idea behind the algorithm and some of the terminology and notation here, using the modern notation of *natural semantics*.[30]

[26] Curry returned to the issue of functionality of types in Combinatory logic in "Combinatory Logic, Volume 1" [1958, Chapter 9], published 24 years later.

[27] Newman was Turing's mentor at Cambridge and the head of the "Newmanry" group at Bletchley Park that created the Colossus code-breaking computer.

[28] MacQueen [2015] gives an explanation of how Newman's algorithm worked – it was not based on type equations.

[29] Robinson's algorithm solves a set of equations between type terms involving type metavariables. The solution takes the form of a substitution (a mapping from type metavariables to types), which can be applied across the entire derivation. Robinson is not cited by Curry, so he appears to have independently discovered the algorithm.

[30] The term "natural semantics" was coined by Gilles Kahn [Kahn 1987] to describe an inference-rule based style of semantics that mimicked the sequent style of natural deduction [Ryan and Sadler 1992, Section 4.3].

We start with *typing judgements*. In their simplest form these are assertions that a given term M has a given type T, written "$\vdash M \Rightarrow T$,"[31] where M is an term in some syntax for value expressions and T is an expression in some type syntax. Let us assume that the term language is a λ-calculus with some basic constants and primitive operators, and type terms are formed from type variables (e.g., t) and primitive type constants (e.g., `int` and `bool`) combined with a right-associative, binary operator representing function types ($\rightarrow$). We start with some assumed judgements such as

$$\vdash \texttt{0} \Rightarrow \texttt{int} \qquad\qquad \vdash \texttt{true} \Rightarrow \texttt{bool}$$
$$\vdash \texttt{+} \Rightarrow \texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{int} \qquad\qquad \vdash \texttt{not} \Rightarrow \texttt{bool} \rightarrow \texttt{bool}$$

We can infer additional typing judgements using *inference rules*, one of the most basic being the rule for application:

$$\frac{\vdash M_1 \Rightarrow T_1 \rightarrow T_2 \qquad \vdash M_2 \Rightarrow T_1}{\vdash M_1\, M_2 \Rightarrow T_2} \qquad \textsc{(App)}$$

which simply says that if a function expression of type $T_1 \rightarrow T_2$ is applied to an argument expression of type T_1, the result of the application has type T_2. This simple form of typing judgements suffices if we are only dealing with closed terms (with no free variables), but the inference rule for typing λ abstractions requires a richer notion of judgement that works for terms with free variables. Thus a context must be provided to give the types of any free variables, yielding the following form of judgement:

$$\Gamma \vdash M \Rightarrow T$$

where Γ is a typing environment — a finite map from term variables to type expressions. We can also think of Γ as a finite sequence of typing assertions like $x : int, y : bool$. It is assumed that the domain of Γ contains all the free variables in the term M. Then the inference rule for λ abstraction is as follows:

$$\frac{\Gamma, x : T_1 \vdash M \Rightarrow T_2}{\Gamma \vdash \lambda x.M \Rightarrow T_1 \rightarrow T_2} \qquad \textsc{(Abs)}$$

This rules says that if, when we assume that x has type T_1, M has the type T_2, then $\lambda x.M$ can be assigned the type $T_1 \rightarrow T_2$, and we can then drop the variable x from the context because it is not a free variable of the abstraction $\lambda x.M$. In these rules, the symbols T_1 and T_2 are *metavariables* ranging over arbitrary type expressions, so in fact the inference rules are rule *schemas*, and there are infinitely many instance rules obtained by substituting particular type expressions for the type metavariables. But we can also substitute generalized type terms containing type metavariables for the metavariables!

These rule instances plug together like Lego bricks to form *type derivations* (i.e., "proofs" of typing judgements), but getting the interfaces between the constituent rule instances to match up requires that certain constraints be satisfied. For instance, let us construct a derivation of a typing for the term $(\lambda x.x)$ `true`.

$$\frac{\dfrac{x : T \vdash x \Rightarrow T}{\vdash \lambda x.x \Rightarrow T \rightarrow T} \qquad \vdash \texttt{true} \Rightarrow \texttt{bool}}{\vdash (\lambda x.x)\,\texttt{true} \Rightarrow T}$$

[31] There are many different syntaxes for these judgements; we follow the notation used in the Definition of Standard ML [Milner et al. 1990].

In order for these rule instances to fit together, it is necessary that $T = \texttt{bool}$, which can be solved by the substitution $\{T \mapsto \texttt{bool}\}$. Applying this substitution to the entire derivation (including contexts) gives

$$\dfrac{\dfrac{x : \texttt{bool} \vdash x \Rightarrow \texttt{bool}}{\vdash \lambda x.x \Rightarrow \texttt{bool} \to \texttt{bool}} \qquad \vdash \texttt{true} \Rightarrow \texttt{bool}}{\vdash (\lambda x.x)\,\texttt{true} \Rightarrow \texttt{bool}}$$

Also note that, as illustrated by this example, the rules are syntax-driven, in the sense that there is one rule governing each form of term construction (abstraction, application, etc.). So for any term we can construct the template of its typing derivation, which will be analogous in structure to the term itself, and to make the rules "match," we have to solve a set of equations between type terms involving type metavariables. As mentioned above, Robinson's unification algorithm [1965] provides a way to solve such sets of equations producing the necessary substitutions.

Sometimes after solving the equational constraints in a derivation and applying the resulting substitution, there remain some residual type metavariables that were not eliminated. An example would be

$$\dfrac{\dfrac{x : T_1 \vdash x \Rightarrow T_1}{\vdash \lambda x.x \Rightarrow T_1 \to T_1} \qquad \dfrac{y : T_2 \vdash y \Rightarrow T_2}{\vdash \lambda y.y \Rightarrow T_2 \to T_2}}{\vdash (\lambda x.x)\,(\lambda y.y) \Rightarrow T_1}$$

with the constraint that $T_1 = T_2 \to T_2$, yielding the substitution $\{T_1 \mapsto T_2 \to T_2\}$. Applying the substitution to the derivation yields

$$\dfrac{\dfrac{x : T_2 \to T_2 \vdash x \Rightarrow T_2 \to T_2}{\vdash \lambda x.x \Rightarrow (T_2 \to T_2) \to (T_2 \to T_2)} \qquad \dfrac{y : T_2 \vdash y \Rightarrow T_2}{\vdash \lambda y.y \Rightarrow T_2 \to T_2}}{\vdash (\lambda x.x)\,(\lambda y.y) \Rightarrow T_2 \to T_2}$$

The residual metavariable T_2 that was not eliminated can be replaced by any type at all in the derivation, so the term $(\lambda x.x)(\lambda y.y)$ can have *any* instantiation of the type scheme $T_2 \to T_2$ as its type (e.g., **int** $\to$ **int**). We capture this multiplicity of types by saying the term has the *polymorphic* type (or *polytype*) $\forall\alpha.\alpha \to \alpha$.

If the equational constraints arising from a schematic derivation are not solvable (e.g., they include an equation like **int** = **bool** or $T = T \to T$), then a type error has been detected.

4.2 Milner's Type Inference Algorithm

When Milner began his project to develop the Edinburgh LCF system, his plan was to replace Lisp, the metalanguage in the first generation Stanford LCF system, with a *logically secure* metalanguage, where a bug in the metalanguage code could not lead to an invalid proof. He chose Landin's ISWIM language [1966b] as the basic framework for the metalanguage, but because ISWIM was dynamically typed it would not provide the desired security. So Milner added a type system to ISWIM, among other features, to guarantee that logical formulas would only be blessed as theorems if they were valid in the LCF logic (i.e., were produced as the conclusions of proofs). The simplest solution would have been to use a simply typed variant of ISWIM, since ISWIM was a *syntactically sugared* version of the untyped λ-calculus, but that choice would have sacrificed much of the flexibility and expressive power that he was accustomed to with Lisp. At that time, Milner was not aware of the previous work on type inference by Newman, Curry, and Hindley, so he invented his own version of type inference with one very significant twist. Instead of a whole-program (or whole-expression) approach to the type assignment problem, he added a special treatment for let bindings (local definitions), such that the definiens subexpression in a let expression could

be assigned a typescheme (a *polymorphic* type) and the defined variable(s) could then be used "generically" in the body of the let-expression (i.e., different occurrences of the variable could be assigned different instances of its polymorphic type).

Milner implemented this typing algorithm in the ML type checker in the Edinburgh LCF system and later published a theoretical development of what he called *Algorithm W* in the paper "A theory of type polymorphism in programming" [1978]. The term polymorphism, used in this sense, had been in common use since Christopher Strachey introduced the term *parametric polymorphism* in his 1967 lectures on "Fundamental Concepts of Programming Languages" [1967].

Let us review the type checking regime that Milner set out in his 1978 paper. He worked with a simple variant of ISWIM with `let` and `letrec` expressions for local definitions of ordinary variables and recursive functions, respectively. The language also provides some primitive types like `int` and `bool` and some basic data structures like pairs and lists with their associated polymorphic operations (e.g., $\texttt{fst} : \forall(\alpha, \beta).\, \alpha \times \beta \to \alpha$).

Let-bindings are treated specially by potentially assigning the defined variable a polymorphic type. In an expression of the form "`let x =` e_1 `in` e_2," the definiens e_1 is typed as above in a context derived from any enclosing definitions or λ-bindings that determine the types of the free variables that appear in e_1. The type derived for e_1 is assigned to the bound variable `x`, and any remaining type metavariables in that type are converted to *generic* (i.e., quantified) type variables – *but only if they do not appear in the types of any λ-bound variables in the context.* For example, consider the following two function definitions:

```
let g = λx. let f = λy. (y,x) in (f 2, f true)
let h = λx. let f = λy. cons(y,x) in (f 2, f true)
```

In the first definition, the λ-bound parameter `x` will be assigned a type metavariable T_x, representing its initially unknown type, and the local let binding of `f` will be assigned a polymorphic type $\alpha \to \alpha \times T_x$, where α is generic (i.e., $\texttt{f} : \forall\alpha.\, \alpha \to \alpha \times T_x$). Thus the body will type check because the type of f can be instantiated to both $\texttt{f} : \texttt{int} \to \texttt{int} \times T_x$ in `f 2` and $\texttt{f} : \texttt{bool} \to \texttt{bool} \times T_x$ in `f true`. The type of definiens for `g` will be $T_x \to (\texttt{int} \times T_x) \times (\texttt{bool} \times T_x)$, and since the context is empty, T_x can be "generalized" (turned into a generic, or quantified, type variable), giving $\texttt{g} : \forall\alpha.\, \alpha \to (\texttt{int} \times \alpha) \times (\texttt{bool} \times \alpha)$.

In the second definition, the λ-bound parameter `x` will again initially be assigned a type T_x, and the inner parameter `y` will be assigned type T_y. Then, to type check the body `cons(y,x)`, we are forced to equate T_x with T_y `list`, which means that T_y `list` replaces T_x everywhere, including in the type of `x` in the context. So now the type of `x` is T_y `list`, which means that T_y occurs the type of the λ-bound variable `x` in the context of the let binding of `f` and thus it is not generic (i.e., not replaced by a quantified type variable) in the type of $\texttt{f} : T_y \to T_y\ \texttt{list}$. Since T_y is not generic, it cannot be instantiated differently in the two expressions `f 2` and `f true`. Assuming `f 2` is typed first, T_y will be replaced (everywhere) by `int`, making the type of $\texttt{f} : \texttt{int} \to \texttt{int list}$ and then the typing of `f true` will fail.[32]

Note that the types of λ-bound variables are never generic. In other words, the argument of a function cannot have a polymorphic type. As a consequence of this, polymorphic types assigned to variables are always in *prenex* form, with all type variable quantifiers at the outermost level; thus a type such as $(\forall\alpha.\, \alpha\ \texttt{list} \to \texttt{int}) \to \texttt{int}$ with inner quantifiers will never occur.

[32]Type metavariables like T_x are not valid forms of type expressions; they are temporary placeholders (representing unknown types) used in the type inference algorithm and they must be eliminated either by substitution or by being "generalized" and turned into universally quantified type variables. Thus they are sometimes called "unification variables." Milner's description did not notationally differentiate type metavariables and quantified type variables — they were both represented by lower-case Greek letters.

In Section 3 of the 1978 paper, Milner provided a denotational semantics for both value terms and type terms, and a declarative definition of the well-typing of an expression (with a context). He then proved a *Semantic Soundness* theorem that intuitively says that the value of a well typed expression belongs to the denotation of its type. The semantic evaluation function was defined to produce a special value *wrong* if a semantic type error is detected (e.g., when applying a non-function to an argument), where *wrong* is not a member of the denotation of any type. Thus if an evaluation of a term produces a member of the denotation of its type, the evaluation cannot have gone "wrong" (i.e., produced the value *wrong*).[33]

The next section of the paper presented two versions of a type checking/inference algorithm. The first, called *Algorithm W*, is purely applicative. It is rather inefficient and cumbersome because it involves a lot of machinery to pass around and compose the type substitutions that result from local unifications of type equations as they appear during a traversal of the term being typed. A second, imperative, version is called *Algorithm J*, where a global variable is used to accumulate the substitutions.

Finally, the paper considered several extensions of the subject language. The first was adding some additional primitive data structures, such as Cartesian product, disjoint sum, and lists, with their associated polymorphic primitive operators. This extension is straightforward.

The second extension was to add assignable variables by introducing a `letref` expression (as in LCF/ML). Done naïvely, this feature leads to unsoundness, for instance if `y` is assigned the generic type α `list` in:

```
letref y = nil in (y := cons(2,y), y := cons(true,y))
```

This top-level unsoundness was avoided by requiring the bound variable to have a monotype. We discuss how the issue is dealt with in Standard ML below in Section 4.4.

The third extension was adding user-defined types (and n-ary type constructors) including recursive types, defined using the `abstype` and `absrectype` declaration forms in LCF/ML. The extension of the well-typing algorithms was asserted to be fairly straightforward (as known from their implementation in LCF/ML).

The last two extensions considered were type coercions and overloaded operators. Milner argued that adding coercions to correct some type errors (e.g., using an integer where a real is expected) should be easy to support. The discussion of overloading is even more speculative, with the conclusion that "there appears to be a good possibility of superposing this feature upon our discipline."

Although Milner's paper included a soundness proof for his polymorphic type system for a purely applicative language, there were some theoretical loose ends left unresolved. Luis Damas addressed these issues in his PhD thesis [1984] and a paper on "Principal type-schemes for functional programs" with Milner that appeared in POPL in 1992 [1982]. He proved that the type inference algorithm in [Milner 1978] was sound and *complete*, meaning that if an expression had a typing according to the type inference rules, then algorithm W would produce a type for the expression and that that type would be *principal*, i.e., as general as possible. He also considered two extensions of the basic ML type system: (1) overloaded operators and (2) polymorphic references (see Section 4.4 below).

We next note several techniques that most real implementations of type inference use to improve efficiency or usability of the ML type system.

[33] This approach to semantic soundness has the disadvantage that there is no way to know that all of the necessary checks for "type errors" have been included in the rules defining evaluation. A more modern approach to soundness, now usually referred to as "safety," is based on showing that a transition-style dynamic semantics will not get "stuck" for well-typed terms [Wright and Felleisen 1991] (see Pierce [2002, Section 8.3] for a tutorial explanation).

Manipulating and composing substitutions is quite expensive. One technique for avoiding them is to represent type metavariables as reference cells (a dynamically allocated cell that can be assigned to). Then a substitution is realized by updating that cell with an instantiation; i.e., a link to the type substituted for the type variable. This can lead to chains of instantiations, which can be compressed in an amortized way when traversing types representations. In the implementation of LCF/ML, type metavariables were represented by Lisp symbolic atoms, and substitution was performed by setting a property (called @INSTANCE) of such an atom.

Determining whether a type variable is generic (polymorphic) in the naïve algorithm involves an expensive search through all the types in the context type environment. This can be avoided by annotating type variables with the λ-nesting depth at which the type variable was created. Instantiating a variable during unification propagates this λ-level to type variables in the instantiating type and determining whether a type metavariable is generalizable then requires only comparing its internal λ-nesting depth with the current λ-nesting depth [Kuan and MacQueen 2007]. An alternate form of binding-level bookkeeping due to Didier Rémy, and used in the Caml family of compilers, uses let-binding nesting levels instead of λ-binding nesting levels [Rémy 1992].

Milner's Algorithm W uses a top-down, left-to-right traversal of the abstract syntax tree of the term being typed. In fact, there is quite a bit of flexibility in the choice of traversal order, and some traversal orders provide better locality information when a type error is detected. For instance, Oukseh Lee and Kwangkeun Yi [1998] propose an alternate algorithm M that uses a bottom-up order. In practice, the type checkers used in compilers often use a mixture of top-down and bottom up traversal, mainly to improve locality information about type errors (see Section 4.7 below).

There has also been considerable theoretical analysis of the type inference problem as process of collecting constraints and then solving them. See Pottier and Rémy's "The Essence of ML Type Inference" [2005] for a summary of this work.

There is, finally, a fundamental principle underlying parametric polymorphism that virtually all implementations depend on: the principle of *uniform representation*. The algorithm invoked by a polymorphic function is expected to work the same no matter how the polymorphic type of the function is instantiated. For instance, the list reversal function "`rev : 'a list -> 'a list`" performs the same algorithm whether it is reversing a list of integers, booleans, string lists, or functions. The algorithm works only on the *list* structure and is insensitive to the nature (or type) of the elements of the list. In order to support this principle, implementations use a uniform representation for all values, typically a 32-bit or 64-bit "word." Small values like booleans or small integers are represented directly in the content of the word, while larger and more complex values like a function are represented by pointers to a heap-allocated structure. Often a tag bit is used to differentiate between a direct value ("unboxed") and a pointer value ("boxed").

4.3 Type Constructors

Types as expressions can be atomic, like the primitive type `int` or a type variable α, or they can be compound, like `int` $\rightarrow$ `bool`. Compound type expressions are constructed by applying *type constructors* to argument type expressions. Type constructors are characterized by having an *arity* or *kind*; thus "$\rightarrow$" is a type constructor of arity 2 or kind $Type \# Type \Rightarrow Type$, i.e., it is a binary operator on types. We can treat primitive types like `int` as type constructors of arity 0.[34]

4.3.1 Primitive Type Constructors. The Curry-style type assignment system studied by Curry and Hindley had just one primitive binary operator on types, "$\rightarrow$," for constructing function types. The

[34]In some formal developments, some forms of compound type expressions are also called "constructors" and are distinguished from "types." We are taking a more naïve view here.

ML type system is much richer. It has this function type operator and many others, and new type operators can be defined by programmers.

For product types, LCF/ML had a single binary infix operator "#," with product values (pairs) constructed by the comma operator. Tuples of length greater than two were approximated by nested (right associative) pairing. Cardelli ML kept the binary product operator but added a labeled record construct with selectors. During the Standard ML design, there was a fairly early consensus on including records, but there were multiple proposals for record syntax and semantics and the design took a long time to settle. In the end, the final form of records used a class of field label identifiers, with types such as

```
{a: int, b: bool, c: string}
```

and corresponding record expressions such as

```
val r = {c = "xyz", a = z-2, b = not p}
```

The order of fields in the type was not important (canonically the fields would be ordered lexicographically), but the order that the fields were written in a record expression determined the order in which they would be evaluated. Selector operations would be automatically generated from field labels; e.g., `#a r`. Record labels can be reused in different record types, so the meaning and type of a selector operator like `#a` will depend on the type of its argument, and it is an error if its type cannot be statically resolved.

Record patterns were also supported, with the ability to specify partial record patterns using "`...`" to represent the unspecified part of the record (e.g., `{a = x, ...}`). The exact type of the record, however, must be inferable from the context.

The binary pairing operator of LCF/ML was dropped in SML, but ordered tuples could be represented by records with natural number labels (successive from 1 to n, for some n, with no gaps). The normal, abbreviated syntax for a tuple expression (in this case a 3-tuple) is (e_1, e_2, e_3). The tuple expression is evaluated by evaluating the constituent element expressions from left to right and returning the tuple of the resulting values. The type of a tuple expression is an n-ary product type, written $t_1 * t_2 * t_3$, where t_i is the type of the corresponding element expression e_i. Tuples are heterogeneous in the sense that different element types can be mixed and they are "flat" in the sense that a 3-tuple is not formed by nested pairing. The selector function for the nth element of a tuple t where $1 \leq n \leq \text{length}(t)$, is `#`$n$. For example, the expression `#2(1,3, true)` returns `3`.[35]

4.3.2 Type Abbreviations. The simplest way for a programmer to define a new type or type operator is to declare a so-called *type abbreviation*.[36] Here are two examples:

```
type point = real * real
type 'a pair = 'a * 'a
```

In the scope of the first declaration, `point` is simply an abbreviation for the type expression `real * real`. And `pair` is a simple type function such that `int pair` expands to (reduces to) `int * int`. So in this example, the types `point` and `real pair` are the same. Type abbreviations cannot be recursive or mutually recursive, so it is always possible to eliminate type abbreviations by expanding their definition, which is, in principle, what happens during type checking. Type abbreviations do not add complexity to the type checking algorithm since they could, in principle, be eliminated before type checking.

[35] By the way, the generic *length* function over tuples is not expressible in SML. There is no type containing all tuples.
[36] Type abbreviations were supported in LCF/ML by the **`lettype`** declaration.

4.3.3 Datatypes. Cardelli ML had a notion of "variants" or labeled unions, but since it was decided more or less from the beginning to adopt some form of the datatypes from the HOPE language, Cardelli's labeled unions would have been redundant. Instead, the **datatype** declaration was adopted. Datatypes are a form of type (or type operator) definition that combine the following features in a single mechanism:

- Discriminated union types, with data constructors serving as tags to discriminate between variants.
- Recursive types could only be defined as datatypes. Here the constructors serve as winding/unwinding coercions in an iso-recursive form of recursive type.[37]
- Datatypes can have parameters, and so can define type operators (such as `list`) as well as simple types.

The data constructors of a datatype can both construct values of the type, injecting variant values into the union, and they can also discriminate on and deconstruct datatype values when used in pattern matching.

Landin's ISWIM had an informal style of data type specification that was not part of the language itself, and was therefore not statically checkable [1966b].[38] Here is an example of one of these informal type definitions, for an abstract syntax for applicative expressions (λ expressions):

```
An expression is either simple
  and is either an identifier
          or a lambda expression
             and has a bv which is a bound variable
             and a body which is an expression,
or compound
  and has a rator
  and a rand, both of which are expressions
```

This stylized prose description is made of unions and products, with the union variants identified by words like *simple* and *compound* and products (or records) having fields named *bv* and *body* (for a *lambda expression*) or *rator* and *rand* for the fields of a *compound* expression, while *identifier* stands for some previously defined type. The convention is that the names *simple* and *compound* designate recognizer functions that can distinguish between the two variants of the union type `expression`, and similarly for the two variants of *simple* expressions. Meanwhile, *bv*, *body*, *rator*, and *rand* serve as selectors from their respective record types. There would also be a convention for variant names like *simple* and *compound* to be used (perhaps with a standard prefix like "cons" or "mk") as constructors for the corresponding versions of expression, so the union types were implicitly discriminated unions.

In Standard ML, we can express this abstract syntax using a pair of mutually recursive datatypes.

```
datatype simple =
  = MkIdentifier of identifier
  | MkLambdaExpression of {bv: variable, body: expression}
```

[37] The term *iso-recursive* indicates a form of type recursion that is mediated by coercive operations relating the recursive type to its unfolded form. An alternative is *equi-recursive* types, where the recursive type is directly equivalent to its unfolding. See Pierce [2002, Section 20.2] for further discussion.

[38] Landin's style of definition was probably inspired by McCarthy's earlier system for defining the abstract syntax of languages [McCarthy 1963, 1966].

```
and expression =
  = MkSimple of simple
  | MkCompound of {rator: expression, rand: expression}
```

and the definition in HOPE would be a minor notational variant of this.

A bridge between Landin's informal method of defining structured data and the datatypes of HOPE and Standard ML was provided by a simple language, named *NPL*, that Burstall described in a 1977 paper [1977]. *NPL* had a form of types with data constructors where the type and its constructors could be declared independently, as with Standard ML's primitive `exn` type. This language had clausal, first-order, function definitions and patterns built with data constructors.

One important change that HOPE introduced relative to NPL was to require "closed" datatype declarations, where a datatype and all its constructors were declared together. The point of this change was to allow optimized runtime representations of constructors and optimized pattern matching, where a set of patterns (for a clausal function or case expressions) could be processed to produce very efficient matching code with no backtracking [Baudinet and MacQueen 1985; Maranget 2008; Pettersson 1992; Sestoft 1996]. The second of these optimizations was part of the first HOPE compiler and was refined in the SML/NJ compiler.

Datatype declarations are *generative*, in the sense that whenever a datatype is statically processed (we say "elaborated") a fresh type constructor is created that is distinct from all existing type constructors. The rationales for this *generative* form of type declaration were:

- It is consistent with the way that abstract types (declared with **`abstype`** and **`absrectype`**) worked in LCF/ML and datatypes in HOPE. Datatypes in part replaced the **`abstype`** mechanism of LCF/ML, where they were the only way to define recursive structured types like trees.
- It made comparison of types in the type checker simple, since **`datatype`** (and **`abstype`**) constructors behaved as atomic, uninterpreted operators in type expressions.
- It avoided a serious technical problem that would have arisen if type equivalence treated recursive types *transparently*; i.e., adopted equi-recursive types instead of iso-recursive types (which datatypes embody). Marvin Solomon [1978] had shown that the type equivalence problem with n-ary ($n > 0$) recursive type operators was the same as the equivalence of deterministic push-down automata (which was not known to be decidable at that time). Much later this problem was determined to be decidable, but the decision algorithm was not simple or efficient [Pottier 2011].

Datatypes are allowed to occur within a dynamic context, such as an expression or even a function body, but such a datatype still has a unique *static* identity.[39] But if a datatype occurs within a functor body, it creates a new static type constructor for each application of the functor. Thus both datatype declarations and functor applications (if the functor body defined datatypes) have a *static effect* of generating new types, or rather, new unique type constructors.

SML has a couple of peculiar pseudo-datatypes: `ref` and `exn`. The primitive `ref` type is treated as if it was defined by

```
datatype 'a ref = ref of 'a
```

in the sense that the `ref` data constructor may be used like a normal constructor to both construct values in expressions and destruct values in patterns, but `ref` is not really a datatype, since its values are mutable, with special equality semantics.

[39] This design decision was rather odd, and it is not considered good programming practice to define types within expressions. On the other hand, some have advocated for module declarations within expressions, which would entail the same thing.

The `exn` constructor, on the other hand, is a very convenient way of simplifying the handling of multiple potential exceptions by a form of pattern-matching similar to a case expression. The `exn` type is effectively a special, open-ended datatype whose constructors (exception constructors) can be declared separately, with no bound on the number of constructors. Thus the `exn` type can be used (or abused?) in cases where we need an "open-ended" union, although this feature was not the original intent, and there is no way to create new open datatypes like `exn`.

4.4 The Problem of Effects

The type inference schemes of Newman, Curry, Hindley, and finally Milner with Algorithm W had the beautiful property of producing the *principal* or most general typing for a term. For ML, however, there was a problem: the problem of side-effects, which, if not handled with great care, could make the type system unsound.

This problem was manifested in LCF/ML in terms of the **`letref`** declaration, which introduced an initialized assignable variable. This construct could potentially allow unsound examples such as:

```
letref x = [] in
   (x := [1];
    not (hd x)) ;;
```

If `x` is assigned the polymorphic type $\forall\alpha.\,\alpha$`list` then the two occurrences of `x` in the body of the let-expression could be assigned two independent instances of that polymorphic type, namely `int list` and `bool list`, and the expression would type check with type `bool`, thus transforming the value `1` into a boolean! To avoid this problem, LCF/ML introduced the following rather obscure and *ad hoc* constraints on the typing of variable occurrences (where "\" represents λ) [Gordon et al. 1979, Page 49]:

(i) If x is bound by \ or **`letref`**, then x is ascribed the same type as its binding occurrence. In the case of **`letref`**, this must be monotype if (a) the letref is top-level or (b) an assignment to x occurs within a \-expression within its scope.

(ii) If x is bound by **`let`** or **`letrec`**, then x has type ty, where ty is an instance of the type of the binding occurrence of x (i.e. the *generic* type of x), in which type variables occurring in the types of current \-bound or **`letref`** bound identifiers are not instantiated.

In the example above, the `letref`-bound `x` could not be assigned a polymorphic type because the expression was (assumed to be) at top-level. Thus the assignment statement `x := [1]` would force x to have the monotype `int list` and the expression `not(hd x)` would cause a type error because `(hd x) : int`.

The next step toward a sound treatment of assignment and polymorphism occurred in the early 1980s. In 1979, Gordon wrote a brief note "Locations as first class objects in ML" [1980] proposing the `ref` type for ML, with a restricted type discipline he called *weak polymorphism* based on *weak type variables*. Gordon suggested this as a research topic to Luis Damas, who eventually addressed the problem in his PhD thesis [Damas 1984, Chapter 3] using a rather complex method where typing judgements were decorated by sets of types involved in refs. Cardelli got the idea to use the ref type either from Gordon's note or via discussions with Damas, with whom he shared a grad-student office for a time. At the end of Chapter 3 of his thesis, Damas returned to a simpler approach to the problem of refs and polymorphism similar to the weak polymorphism suggested by Gordon, and he mentioned that both he and Cardelli implemented this approach.[40] Weak polymorphism is not

[40] At one point, Cardelli had Damas write down a rough, half-page sketch of his algorithm that was circulated for several years and served as the starting point of multiple implementations of the idea. For instance, MacQueen adopted the design in the original type checker for the Standard ML of New Jersey compiler [MacQueen 1983a].

mentioned in the various versions of Cardelli's ML manuals [1982d; 1983b; 1984b], however, where the `ref` operator is described as having a normal polymorphic type.[41]

Now let us illustrate again the need for a careful treatment of polymorphism and assigment when using `ref` types. Assume, as in Cardelli ML and Standard ML, assignment side effects are introduced through the `ref` primitive type and its associated operators. Assume also, that we, naïvely, assign the following ordinary polymorphic types to those operators:

```
val ref : 'a -> 'a ref
val := : 'a ref * 'a -> unit
val ! : 'a ref -> 'a
```

Now with these types, the above LCF/ML example can be coded as follows:

```
let val x = ref []
    in
      x := [1] ;
      not (hd (!x))
    end
```

The problem is that at the binding introducing variable `x`, if `x` is assigned the polymorphic type "`'a list ref`", then the two occurrences of `x` in the body of the let expression can have their types be independently instantiated to "`int list ref`" and "`bool list ref`", which again means that the result would have type `bool` and the integer value `1` will have been converted to a boolean.

The question is how to modify polymorphic type inference for pure expressions to prevent this unsound outcome? The original solution in Standard ML was to introduce a special variety of type variable, called an *imperative* type variables, which are restricted on when they can be generalized. In this example, the primitive function `ref` is defined to have a restricted form of polymorphic type:

```
val ref : '_a -> '_a ref
```

where "`'_a`" is an imperative type variable. Such a type variable can be instantiated to any type τ as before, but the property of being imperative is inherited by any type variables in τ. In our example, the type inferred for "`ref []`" will be "$\_T$ `list ref`," where the type metavariable $\_T$ has inherited the "weak" attribute from the type of `ref`. Now a special restriction controls whether the type of `x` can be generalized to make it polymorphic. Generalization is only allowed if the definiens `ref []` of the declaration of `x` is *non-expansive*, meaning that it is of a restricted class of expressions that do not invoke any "serious" computation. But in this case, the application `ref []` *is* expansive, because it involves the allocation and initialization of a new ref cell. Hence the type of `x` will be ungeneralized $\_T$ `list`. The type metavariable $\_T$ will be instantiated to `int` when type checking the assignment "`x := [1]`," causing the type of `x` to become "`int list ref`." Then the expression "`not (hd(!x))`" will fail to type check.

The theory for imperative type variables is developed in Tofte's paper "Type Inference for Polymorphic References" [1990]. Harper developed a much simpler proof of soundness in [Harper 1994].

This solution using imperative type variables works, but it has some unpleasant side effects. For instance, an alternate definition of the identity function of the form

```
val id = fn x => !(ref x)
```

which temporarily creates a ref cell will be assigned an imperative polymorphic type

```
id : '_a -> '_a
```

[41] The notes at the end of [Cardelli 1982c], mention that the 5-11-82 Unix version has a "New typechecker for ref types." It is not known whether this type checker used weak polymorphism, but it is likely.

even though its use of a reference cell is benign because it is encapsulated and the ephemeral reference cell cannot "escape" to cause problems.

Another issue is that an imperative polymorphic type cannot be allowed to match a pure (that is, ordinary) polymorphic type in a signature specification, so purely internal and contained use of references can contaminate an implementation so that it cannot be used to implement a signature with pure polymorphic value specifications. For example, the function `id` defined above could not be used to satisfy a signature specification of the form "`type id : 'a -> 'a`," which is a kind of violation of abstraction.

In the SML/NJ compiler [Appel and MacQueen 1991; Appel and MacQueen 1987], a slightly liberalized variant of this scheme was implemented whereby imperative type variables had an associated numeric *weakness* and only type variables of weakness 0 were prevented from being generalized. This approach moderately increased the flexibility of the type system, but did not really eliminate the basic problems with imperative polymorphism.

Many people worked on the problem of polymorphic references over a period of several years [Harper 1994; Leroy 1993; MacQueen 1983a; Mitchell and Viswanathan 1996; Tofte 1987, 1988, 1990; Wright 1995, 1992, 1993]. Ultimately, Andrew Wright cut the Gordian knot by proposing to get rid of imperative type variables and replace them with a stronger, but simpler, restriction on the generalization of types in **`let`** expressions [1995]. The key insight is that polymorphic generalization in a pure, call-by-name language is justified by the interpretation of **`let`**-expressions as β-redexes that can be reduced to an equivalent expression by substituting the definiens for all occurrences of the defined variable in the scope of the binding.

$$\textbf{let}\ x\ =\ e_1\ \textbf{in}\ e_2\ \longrightarrow\ [e_2/x]e_1$$

Multiple occurrences of x in e_2 are replaced by multiple copies of the expression e_1. These multiple copies can then be typed in their individual contexts, producing specialized versions of the "generic" or polymorphic type of e_1 that correspond to the different instantiations of that polymorphic type that are produced during the type inference algorithm.

But in an call-by-value language, the dynamic semantics of **`let`** expressions evaluates the definiens once, before substitution. The call-by-name and call-by-value interpretations are not equivalent in the presence of effects, except in the special case that the definiens is a syntactic value (e.g., a variable or λ-expression), which is guaranteed not to produce effects when it is (trivially) evaluated. The failure of type safety resulting from treating references as polymorphic is just one of several manifestation of this discrepancy, as evidenced by the discovery of the incompatibility of unrestricted generalization with first-class continuations [Lillibridge and Harper 1991].

The restriction of polymorphic generalization to syntactic *value* expressions came to be known as the *value restriction*[42] and was adopted in the *Definition (Revised)* because it was found not to be a significant limitation in practice.[43] For example, in the case of a function defined by the partial application of a function

```
val f = g x
```

one can restore polymorphism by η-expansion:

```
fun f y = g x y     or     val f = fn y => g x y
```

[42]In the Definition the restriction is called "non-expansiveness" because of the historical, but inaccurate, association of the problem with the allocation of references.

[43]Wright examined over 200,000 lines of existing SML code and found only 31 η-expansions were required, along with a small number of other minor edits [1995].

which makes `f` a value and, thus, its type will be generalized. The value restriction had some disadvantages, however. For example, η-expansion may delay computations that could be performed once, rather than once on each call, affecting efficiency. Another problem is that η-expansion does not apply to abstract types, such as a type of parsing combinators, in which the underlying representation as functions is not exposed to the programmer. For in that case even simple computations such as the composition of parsers violates the value restriction, precluding polymorphic generalization, and there is no natural work-around. In practice, however, the vast majority of existing SML code was not affected by the switch to the value restriction.

4.5 Equality Types

Equality is normally considered an intuitively "polymorphic" relation. But it technically does not fit the pattern of parametric polymorphic functions, which obey the Uniformity Principle where the underlying algorithm is insensitive to the instantiation of the polymorphic type. In the case of equality, the algorithm for computing equality varies depending on the type of the elements being compared and for some types (e.g., function types) it is not possible to compute equality at all. But it is also the case that equality does not seem to fit the pattern of ad hoc overloaded operators like addition, where the algorithms for different types of arguments may be entirely unrelated.

There is a natural subset of all types consisting of types that *admit equality*. For these types, the computation of equality of values is driven systematically by the structure of the type. Although not *parametric* polymorphic, equality seems to fit a notion of uniform *type-driven* polymorphism. LCF/ML had a rather ill-defined notion of equality [Gordon et al. 1979, Appendix 3]:

> [=] is bound to the expected predicate for an equality test at non-function types, but is necessarily rather weak, and may give surprising results, at function types.

In Standard ML, however, it was decided that a more flexible pseudo-polymorphic treatment of equality should be provided. This was achieved by introducing another special form of polymorphism with another special form of bound type variable, namely *equality type variables*. These type variables arise from uses of the generic equality operation and can be generalized. But when these variables are instantiated, they can only be instantiated to types in a restricted class, namely the types that admit equality based on their known structure. One way of thinking about this class of types is that they are the *hereditarily concrete* types. In particular, function types and abstract types are excluded from this class, as are all types that are built up from function types or abstract types. Recursive datatypes (such as `list`) are included, because they support an inductively defined equality (given that the arguments of the type constructor, if any, support equality).

There are several ways to implement the polymorphic equality operation:

(1) One can use object descriptors that allow a runtime-system function to determine how to compare values for equality. Since this information largely overlaps the requirements for tracing garbage collection algorithms, it may involve little additional cost.
(2) One can implicitly pass an equality operation for the specific instantiation of an equality type variable at runtime to the place where it is needed. This approach is similar to the *dictionary passing* method used to implement Haskell type classes [Wadler and Blott 1989].
(3) One can monomorphize the program, which turns polymorphic equality into monomorphic equality.

For Standard ML, the first implementation strategy is most common, but there have also been examples of the second [Elsman 1998] and third approaches [Cejtin et al. 2000]. Also in cases where an occurrence of equality is assigned a ground type, compilers can produce specialized inline code to implement equality for that particular type, avoiding the need for runtime tracing of structure (dictionary-passing can also be optimized in such cases).

There are a couple of significant problems with polymorphic equality (beyond the additional complexity in the type system). The first of these is that there is a hidden performance (or complexity) trap in polymorphic equality if it is used naïvely. Generic equality of data structures, such as lists and trees, takes time that is proportional to the size of the data structures. In general, when testing equality on complex structures, exhaustive traversals of the structures being compared is not an efficient approach. Often there is some designated component of the complex structure (say, a unique identifier, or reference cell), that can serve as a proxy for the structure with regard to equality and which can be compared efficiently. Another problem is that structural equality is often the wrong notion when comparing values of a data abstraction. For example, a red-black-tree implementation of finite sets admits multiple concrete representations of the same abstract set value. Experienced programmers learn to avoid the use of polymorphic equality because of these problems and instead define an efficient *ad hoc* equality operation for their nontrivial types. Such a specialized equality operation is typically made available as part of the signature of a module, but it has to be invoked via a specialized name because the operator "`=`" denotes only the generic version of equality. For this reason some Standard ML designers (Harper and MacQueen) suggested that polymorphic equality be removed from the language during the work on the *Definition (Revised)*, or at least that it be deprecated.

4.6 Overloading

LCF/ML had no overloading. There was only one numeric type, `int`, and the arithmetic and order operations were defined only for `int`. On the other hand, HOPE had a very general scheme for overloading, where functions, constants, and datatype constructors could all be overloaded. In HOPE, overloading was resolved against the full contextual type, which allowed overloaded constants and overloaded operations that were resolved by their return type. The general overloading scheme coexisted fairly well with polymorphism (borrowed from LCF/ML), but the overloading resolution algorithm was non-trivial, being based on Waltz's relaxation algorithm for computer vision [Waltz 1975].

The complexity of overloading in Hope led some of the SML designers to oppose adding overloading to the language, but, in the end, the decision was to support "conventional" overloading of arithmetic and order operators on numeric types, and order operations on character and string types. Since all the instances of overloaded operators are of ground type, it is fairly easy to resolve overloading, but it must be resolved in a second phase after type inference, because type inference provides the contextual type information needed to resolve overloading. The *Definition (Revised)* extended this mechanism to also support overloading of literals (e.g., integer literals can be overloaded in terms of precision) and added the notion of default types for overloaded constants.[44]

4.7 Understanding Type Errors

Type inference imposes a pragmatic cost: determining the cause of type errors can sometimes be difficult, because type information flows from one place in the program text to another, possibly distant, point by way of a series of unifications and their resulting type variable substitutions. Most type checkers detect an error (depending of course on the details of the algorithm used) at a point where two pieces of type information come into conflict. Because of the implicit type flow, the program point where a conflict is detected may not be the source of the type error. The cause of the error may also not be one of the points where the conflicting pieces of type information originate,

[44]Before default types, the function "`fn x => x+x`" could not be assigned a type, which was quite mysterious to novice users, but under the *Definition (Revised)*, the type of this function would be "`int -> int`," since "`+`" defaults to integer addition.

but some intermediate point where the type flow is misdirected (e.g., by projecting the wrong component of a pair).

In practice, the problem of understanding the cause of type error messages is mainly an issue for relatively novice ML programmers. Experienced programmers develop a facility for mentally tracing the flow of types, and they can easily find the cause of most type errors. The long-distance flow of type information that can lead to mysterious type-error messages can be interrupted by module boundaries and by selective addition of explicit type specifications in declarations, which also help make code more readable. In those rare cases where the cause of a type error is not evident after a few minutes thought, the source of an error can generally be determined by judicious addition of some explicit type specifications.

As mentioned above (Section 4.2), the type checking algorithms used in a compilers have a great deal of freedom to choose the order in which the abstract syntax tree is traversed. Type checkers typically do incremental unifications to solve typing constraints at each node of the abstract syntax tree, but they can use different strategies to control the "timing" of unifications that may produce type errors. This flexibility can be used to improve the locality of error messages by careful choice of the order of traversing the abstract syntax, using a combination of top-down and bottom-up strategies [Lee and Yi 1998], or by choosing the points where unifications are performed [McAdam 2002, 1998]. The default traversal orders used in theoretical descriptions, such as Milner's Algorithm W, were not designed with the goal of good type error messages and are therefore not optimal for this purpose.

Nevertheless, deciphering type error messages is clearly a challenge for students learning ML. Facilities that help explain type errors can also be useful for teaching programmers how type inference and polymorphic typing work, even in the absence of type errors. Thus, over the years, starting with Mitchell Wand [1986] and Gregory Johnson and Janet Walz [1986], over forty publications (including four PhD theses) have proposed solutions to the problem of producing better type error messages that make it easier to identify the source of the error.

Many of these proposals to improve type error messages involve "instrumentation," that is, annotating various structures involved in type checking with type or program location information. Type annotations can be attached to the abstract syntax tree and program locations can be attached to types or type variables or type substitutions during the process of type checking. These annotated structures can then be analyzed at the point of error detection or after the fact to produce more exact and helpful information on the source of type errors [Beaven and Stansifer 1993; Chitil 2001; Choppella 2002; Duggan and Bent 1996; MacQueen 2010; Wand 1986].

Other techniques that have been used in addition to instrumentation, or sometimes instead of instrumentation, include annotated constraint solving [Hage and Heeren 2006; Pavlinovic et al. 2015; Stuckey et al. 2004; Wazny 2006]; program slicing [Haack and Wells 2004; Schilling 2011; Tip and Dinesh 2001]; program modification [Bernstein and Stark 1995; Lerner et al. 2007; Sharrad et al. 2018]; attribute grammars [Johnson and Walz 1986]; Bayesian statistics [Zhang and Myers 2014]; SMT solving [Pavlinovic et al. 2015]; and machine learning [Wu et al. 2017].

5 MODULES

The Standard ML module system was the most novel and technically challenging part of the language. Its origins go back to discussions between Rod Burstall and David MacQueen as early as 1977, when work on the HOPE language was just beginning. MacQueen wrote a proposal to add modules to HOPE in 1981 [1981] and this design eventually became the basis for the early module proposals for Standard ML. The initial design proposal was circulated in August 1983 following the first SML design meeting. A year later the first published module design proposal was presented at the 1984 Lisp and Functional Programming conference [MacQueen 1984], alongside Robin Milner's

proposal for the Core language (the language of expressions and their types). Another final (pre-Definition) version appeared in *Polymorphism* in October 1985 [MacQueen 1985b] and in an LFCS[45] tech report in March 1986 [Harper et al. 1986].

The modules design was initially inspired by previous work in the field of algebraic specification, particularly the work of Burstall and Joseph Goguen on the specification language Clear [1977]. The design was also influenced by basic ideas from universal algebra [Cohn 1965] and category theory, and later on by ideas from Per Martin-Löf on intuitionistic type theory [1982].

The basic principles behind the design were as follows:

- A module is a collection of named components, which can typically be types or values, syntactically expressed by encapsulating a sequence of declarations with the keywords "`struct`" and "`end`," forming a basic *structure expression*.[46]
- A module has a static description or "type," which we call a *signature*.[47] A signature contains a static specification for each component of the module; e.g., a type specification or the type of a variable. Signatures play the same rôle for modules as types do for values, and the signature of a module can be statically checked and inferred, just as types are checked and inferred for expressions.
- Module definitions and signatures are *independent*. Several different signatures can describe (via *ascription*[48]) a given module, and a given signature can be ascribed to different modules. So the relation between modules (implementations) and signatures (interfaces) is many to many, unlike many other module systems, where there is a fixed binding between a particular module and *its* signature.
- Modules, as well as types and values, can be components of other modules. So modules can have a hierarchical structure. This nesting of modules means that signatures may also contain specifications of module components by their name and signature.
- A module expression can be lambda-abstracted with respect to a module name (with an ascribed signature) to form a module function, which is called a *functor*. This is an instance of *Landin's Correspondence Principle* [Landin 1966b; Tennent 1977]. A new module can be constructed by applying a functor to a module argument.[49]
- Functors give rise to a requirement on the module-system design to provide a mechanism for specifying when two type components within the functor parameter are equal. SML achieves this mechanism via *sharing constraints*.
- Module signatures can control access to components selectively. Thus components can be hidden (i.e., not *exported*) simply by not including them in a signature ascribed to a module. Thus signature ascription expresses an interface coercion.

[45] Laboratory for Foundations of Computer Science, University of Edinburgh.
[46] The word "structure" was suggested by the terminology of "mathematical structure" from universal algebra, where it denotes a model of some signature and set of axioms (e.g., a group, or a ring).
[47] Another term from universal algebra.
[48] *Ascription* means asserting that a module *has* or *satisfies* a given signature using the notation "`M : SIG`." This mechanism is the module analogue of the similar notation for ascribing types to expressions; i.e., "`e : ty`."
[49] The term "functor" is borrowed from category theory. An alternate suggestion was "parametric module," which Burstall objected to on the grounds that we don't call ordinary functions "parametric values."

5.1 The Basic Design

As mentioned above, a simple module, or *structure*, is an *encapsulated* sequence of declarations, where later declarations may mention names declared earlier in the sequence. The encapsulation is expressed by the syntax

```
struct
  declarations
end
```

where the keywords `struct` and `end` delimit the declarations forming the module (structure) body. This syntax is the basic form of *structure expression*. Usually, a structure is named in a *structure declaration* of the form:

```
structure S =
  struct
    declarations
  end
```

where the declaration form is introduced by the keyword `structure`, and `S` is the name to which the structure is bound.

A simple example is

```
structure Nat0 =
  struct
    datatype nat = Z | S of nat
    val zero = Z
    fun add Z m = m
      | add (S n) m = S(add n m)
  end
```

In the scope of this declaration, the *structure variable* `Nat0` denotes the structure defined by the structure expression following the equal sign.

By default, all the named components (types and values) defined in the declarations of the structure are accessible via the module variable using the *dot notation*; e.g., `Nat0.nat` denotes the type component of the structure and `Nat0.zero` denotes the zero value. In this example, the visible components are those defined by the sequence of three declarations: the datatype `Nat0.nat`, and its associated data constructors `Nat0.Z` and `Nat0.S`, and the values `Nat0.zero` and `Nat0.add`. A structure can also be thought of as a heterogeneous record containing a mixture of types and values (and structures), but some fields of the record *depend on* other, necessarily earlier fields; e.g., the type of `Nat0.add` is "`Nat0.nat -> Nat0.nat -> Nat0.nat`," so `Nat.add` depends, statically, on `Nat0.nat`.

A type for a structure can be expressed as a *signature*, which gives static specifications for its components. Continuing with the `Nat0` example, a possible signature for this structure would be:

```
sig
  type nat
  val zero : nat
  val add  : nat -> nat -> nat
end
```

and this signature could be assigned a name using a *signature declaration* such as:[50]

[50] A common convention is to use initial caps for structure and functor names, and all caps for signatures, but this convention is not enforced by the language.

```
signature NAT =
  sig
    type nat
    val zero : nat
    val add  : nat -> nat -> nat
  end
```

Once we have a way of defining both structures and signatures, the question arises of how they are related. Though these declarations are entirely independent, meaning neither depends on the other, it will be the case that the structure `Nat0` will *match* the signature `NAT`. The result of that matching can be expressed by the *signature ascription* "`Nat0 : NAT`," which is a well-typed structure expression that has a signature that is different from innate signature of `Nat0`.

To understand why, consider this similar declaration of a structure `Nat1` where the structure variable is constrained by the signature `NAT`:

```
structure Nat1 : NAT =
  struct
    datatype nat = Z | S of nat
    val zero = Z
    fun add Z m = m
      | add (S n) m = S(add n m)
  end
```

The effect of a such a signature constraint in a declaration is defined via a process called *signature matching*, which has two functions:

(1) verifying, by name, that the components specified in the signature exist in the structure and agree with their specification in the signature, and
(2) masking or eliminating components defined in the structure but not specified in the signature.

In the example above, `Nat1` differs from the earlier `Nat0` in that `Nat1.Z` and `Nat1.S` are not available as constructors for `Nat1.nat`, because those components are not specified in the signature `NAT`.

In this particular example, the signature constraint is too restrictive, because the components specified in the signature are sufficient to define only one value of type `Nat1.nat`, namely `Nat1.zero`. Thus, the signature `NAT` is not an adequate description of the interface for the structure `Nat1` or, in other words, the `NAT` signature hides too much of the structure, eliminating essential functionality.

One way of fixing this problem is by using a *datatype specification* in the signature to expose the complete structure of the `nat` datatype. In this example, a more complete signature can be defined as follows:

```
signature NAT1 =
  sig
    datatype nat = Z | S of nat
    val zero : nat
    val add  : nat -> nat -> nat
  end

structure Nat1 : NAT1 = ... as before ...
```

The result is that the datatype specification matches the isomorphic datatype declaration in a structure; i.e., the "algebra" of the datatype specification is isomorphic to the "algebra" of the actual datatype declaration in the matching structure, so that the whole datatype, including its constructors, is exported through the signature.

Another approach to fixing the problem with the NAT signature is to augment the Nat1 structure and its signature with a successor function.

```
signature NAT2 =
  sig
    type nat
    val zero : nat
    val succ : nat -> nat
    val add  : nat -> nat -> nat
  end
structure Nat2 : NAT2 =
  struct
    ...
    val succ = S
    ...
  end
```

Effectively, this solution means that the constructors Z and S are exported indirectly as ordinary (nonconstructor) values zero and succ, and these are sufficient to construct any natural number (e.g., 2 = succ(succ(zero))). But because the data constructors are not available, *as such*, from Nat2, client code cannot use pattern matching over datatype nat to do case analysis and deconstruction of values of that type. We would need to add functions like isZero and pred to recover the full expressiveness provided by the datatype.[51]

Signature matching provides two mechanisms to constrain the components of a structure by a signature.

(1) The signature may specify only a proper subset of the components of the structure. In this case, the structure resulting from the match is *coerced* to have only the components specified in the signature. In this way, a signature constraint prevents access to components of the constrained structure that are not meant to be accessible to its clients.
(2) The type specified for a value component can be a less-general instance of the actual polymorphic type of the component in the structure. For example:

```
signature SIG = sig
    type t
    val id : t -> t
  end

structure S: SIG = struct
    type t = int
    val x = 3      (* a local, nonexported value *)
    fun id x = x   (* type id : 'a -> 'a *)
  end
```

Here the structure S has only two components, "S.t = int" and "S.id : S.t -> S.t" (or "int -> int"). The x component of the structure is hidden by the signature constraint, so S.x is undefined.

[51] This also means that Nat2.nat is a *de facto* abstract type because we cannot access the native structure of the type. On the other hand, it is not properly abstract because it is an equality type, and that attribute is preserved through the signature matching.

This example also illustrates the "transparency" of matching of type components, meaning that the type component `S.t` resulting from matching the specification `type t` is statically identical to the type defined in the right-hand-side structure expression; i.e., "`S.t = int`."

Another major feature of the module system is that it supports the definition of module functions, called *functors*. A simple example is a sorting functor:

```
signature ORD = sig
    type elem
    val le: elem * elem -> bool    (* a less-than-or-equal relation *)
  end

functor Sort (X : ORD) : sig
    val sort : X.elem list -> X.elem list
  end = struct
    fun insert (x, []) = [x]
      | insert (x, y::ys) =
          if X.le(x,y) then x::y::ys else y::insert(x,ys)
    fun sort nil = nil
      | sort (x::xs) = insert(x, sort xs)
  end
```

This functor can be instantiated to sort integers as follows:

```
structure IntOrd : ORD = struct
    type elem = int
    val le = Int.<=
  end
structure IntSort = Sort(IntOrd)
```

and simiarly to sort strings

```
structure StringOrd : ORD = struct
    type elem = string
    val le = String.<=
  end
structure StringSort = Sort(StringOrd)
```

This example brings up an interesting point, which is that while `IntSort.sort` sorts with respect to the usual ordering of integers (`Int.<=`), one can define another sorting structure

```
structure IntRevOrd : ORD = struct
    type elem = int
    val le = Int.>=
  end
structure IntRevSort = Sort(IntRevOrd)
```

that sorts based on the reverse order. The types of `IntSort.sort` and `IntRevSort.sort` are the same, so it is possible to confuse them and use the wrong sorting function in some context. We can make the dependency of the sort function on a particular `ORD` structure explicit by including that structure as a substructure of the result of the `Sort` functor:

```
signature SORT =
sig
  structure Ord : ORD
```

```
    val sort : Ord.elem list -> Ord.elem list
  end

  functor Sort (X : ORD) : SORT =
    struct
      structure Ord = X
      ...
    end
```

But we still need a mechanism to assert that the `Ord` structure for a given sorting structure is the expected one. This requirement leads to the phenomenon of *structure sharing*.

Functors can have multiple structure arguments, which can be considered substructures of a single, anonymous structure argument. These arguments can interact with one another based on common inheritance of types and substructures.[52] For example, suppose that we have the following functor definition:

```
functor F(structure X: SORT structure Y: SORT) = struct ... end
```

where it is important that `X` and `Y` have sort functions with respect to the same ordering function `le` (necessarily over the same type). We can insure this property by adding a *sharing constraint* on the arguments of `F`. This constraint is expressed as

```
functor F(structure X: SORT structure Y: SORT sharing X.Ord = Y.Ord)
  = struct ... end
```

The critical point is that the substructures `X.Ord` and `Y.Ord` have a *static identity* as structures that can be used to *statically* check the sharing specification. This static sharing implies that they are the *same* structure, which ensures that the `le` functions in `X.Ord` and `Y.Ord` agree.

An interesting property of this semantics for "strong" structure sharing is that sharing persists even if structures are coerced to have new interfaces by signature ascription. The underlying identity of a structure is propagated to the structure derived by coercive signature matching, even though the signature of the structure has been changed.

This strong notion of static structure identity was part of the original design and was included in the Definition. But it resulted in some serious complications in the Definition, as noted in the Commentary [Milner and Tofte 1991, Section 6.3], although with somewhat milder impact on implementations of the module system. In other words, while it was not too challenging to implement this sharing semantics in a compiler, it was a significant challenge to express it in the formal Definition.

There is a weaker notion of sharing that does not guarantee that structures are the same (including at the value level), but only that they contain the same static elements (i.e., *types* or *type constructors*) where they overlap. This weaker notion of structure sharing was adopted in the *Definition (Revised)* [Milner et al. 1997] in order to simplify the semantics of sharing. The most important difference between the two forms of sharing is that structure sharing guarantees that any mutable state defined in the shared structures must be the same (since the structures are the same), whereas type sharing does not enforce that guarantee. This weaker form of sharing also does not provide a way to guarantee that the `X.Ord` and `Y.Ord` structures from the example above provide the same ordering function.

[52] The inheritance comes from references to the same components and should not be confused with the notion of class-based inheritance found in object-oriented languages.

5.2 The Evolution of the Design of Modules

As mentioned above, the design of modules for ML was originally inspired by the Clear algebraic specification language of Burstall and Gougen [1977]. The Clear specification language, based on the notion of many-sorted algebraic theories, had a notion of a *signature*, which would specify a set of *sorts* or types, along with a set of *operator symbols* with their types (expressed as sequences of argument and result sorts). A *theory* consisted of a signature together with a set of equations specifying the behavior of the operations. Some of these equations might serve to define the operators, while others might specify properties such as commutativity of an operation.

Clear's major innovation was to provide a set of *theory-building* operations that could build complex specifications out of simpler ones. There were four basic theory building operations: *combine*, *enrich*, *induce*, and *derive*. The meanings of these operations are not relevant here, except to say that all but *induce* (the inductive closure of a theory) have rough analogues in the SML module system.[53]

The power of Clear was greatly extended by the ability to define theory *procedures* (or functions) that express compound theory constructions relative to theory parameters. These theory parameters were constrained by theories that serve as *meta-sorts*, which impose requirements on them, acting roughly as an SML signature if we ignore the rôle of equations. These theory procedures are roughly analogous to functors in the SML module system. There is even a notion of sharing of sub-theories, expressed by identity of names of the sub-theories, which is analogous to sharing in modules.

Over the period 1978 to 1980 MacQueen and Burstall discussed ways to adapt the parameterized theory idea (theory procedures) from Clear to the context of the HOPE programming language that was being designed at that time. These discussions led to a proposal for modules for HOPE that was published at the 1981 Functional Programming and Computer Architecture conference at Aspenäs in Sweden [MacQueen 1981]. The Alphard [Wulf et al. 1976] and CLU [Liskov et al. 1981, 1977] languages, which supported parameterizing over types constrained by associated operation interfaces, were additional influences.

The terminology chosen for HOPE modules differed from Clear in that the analogue of a signature in Clear was called an *interface*, and a theory in Clear roughly corresponded to a *structure* in HOPE, or, to be more precise, a structure corresponds to an algebra that is a *model* of a theory. In the HOPE module proposal, there was no provision for including equations specifying the behavior of operations and constants, only the implementation of operators. Operations could also be higher-order, while Clear, being algebraic, supported only first-order operations.

Interfaces (i.e., signatures) would typically be based on other interfaces. Some examples of interface definitions in the proposed HOPE module system are:

```
interface BOOL
  data bool == true ++ false
end

interface TRIV
  data elem
end

interface POSET(BOOL, TRIV)
  dec < : elem # elem -> bool
end
```

[53] Related, roughly contemporary, investigations of parameterized algebraic specifications were being carried out by the ADJ group at IBM Research and collaborators [Ehrig et al. 1980; Thatcher et al. 1982].

```
interface LIST(BOOL)
  data list a == nil ++ cons(a, list a)
  dec null : list a -> bool
end

interface SORTING (POSET, LIST)
  dec sort : list elem -> list elem
  dec is_ord : list elem -> bool
end
```

Here the POSET interface depends on, and inherits, the types (and operators) from the BOOL and TRIV interfaces, namely the types bool and elem that it uses. Structures implementing (matching) the POSET interface will correspondingly be based on and incorporate structures for BOOL and TRIV. Note that the SORTING interface is based on the interfaces POSET and LIST, both of which are based on BOOL. Implicitly, because of the common mention of the name BOOL, it is assumed that in any structure implementing SORTING, the underlying substructures for POSET and LIST *share* the same BOOL substructure.

The parameters in these interfaces can be instantiated explicitly when defining structures implementing the interfaces and corresponding structure definitions can be parameterized with respect to the required substructures/subinterfaces. Thus, a structure implementing SORTING can be defined in various ways:

(1) `structure Sorting : SORTING(P0,L0)`,
(2) `structure Sorting(P : POSET) : SORTING(P,L0)`,
(3) `structure Sorting(L : LIST)(P : POSET) : SORTING(P,L)`.

where P0 and L0 are the names of particular structures implementing POSET and LIST, respectively. Note also that (2) and (3) are *parameterized structures*[54] and that (3) is also a *curried* definition, implying that a limited form of *higher-order* structure is supported.

Another interesting aspect of this module system design was that datatypes could be specified in interfaces, and the corresponding *free* implementation of such datatypes could then be defined in a matching structure as illustrated in the following code:

```
interface BOOL
  data bool == true ++ false
end

structure Bool : BOOL
  data-rep bool is free
end

structure List : LIST(Bool)
  data-rep list is free
  --- null(nil) <= true
  --- null(cons(x,l)) <= false
end
```

The definition `data bool == true ++ false` implements the type bool as the canonical free algebra over the set of constructors, and similarly for the list type constructor in structure List. Note

[54] Precursors of functors in Standard ML.

also how the LIST interface is applied to the structure argument Bool to explicitly designate what structure instantiates the BOOL interface in LIST. Thus the design supported parameterized signatures and their application to structures.

This proposed module system for HOPE was a paper design and was not implemented, nor was there a metatheory developed for it, but it did provide a stepping-stone on the way to the future Standard ML module system.

In the weeks after the first Standard ML design meeting in Edinburgh (April 1983), there was some discussion of adding a module system to the Bare ML language of Milner's proposal. Because of MacQueen's work on a design for a module system for HOPE, it seemed natural for him to take the lead in this project, and a plan was discussed between Milner, MacQueen and Luca Cardelli during July 1983 by post and at least one phone call. MacQueen produced an outline of a module proposal in mid-July, which was expanded to an initial eight-page proposal, with an additional three pages of explanatory notes, by the beginning of August.

Section 4 of Milner's second draft of the Proposal for Standard ML [Milner 1983c] was concerned with a form of declarations called *directives*. Three varieties of directives were described, the first was to introduce or cancel infix properties for identifiers, the second introduced type abbreviations. The third form of directives, called *specification* directives, was concerned with specifying external type constructor names (with their arity), and value variables, constructors, and exception identifiers (with their types), that are used in a source file but not defined locally. These directives were intended to specify an interface to be satisfied by the context (environment) in which a file is compiled. They were included to support an anticipated separate compilation facility for the language.

MacQueen's initial August 1983 draft proposal took an informal approach, viewing modules as declarations (possibly contained in a separate source file) that require a context environment satisfying appropriate specifications for the declaration's free identifiers (a so-called *inward interface*). The declaration, once compiled and executed relative to this context environment, produces an output environment binding the identifiers declared in the declaration, while static analysis (i.e., type checking) of the declaration would produce a corresponding *outward interface*. Thus a declaration, accompanied by an inward specification covering its free identifiers, was viewed as defining a mapping over *typed* environments, where the environment types are specifications:

$$[\![\langle \mathit{I\text{-}spec}, \mathit{dec}\rangle]\!] \;:\; \mathit{Env}_{\,\mathit{I\text{-}spec}} \;\longrightarrow\; \mathit{Env}_{\,\mathit{O\text{-}spec}}$$

The proposal introduced no specialized module syntax, viewing modules essentially as (potentially separately compiled) source files containing declarations together with their required *inward* specifications to make them self-contained. The environment produced by "evaluating" such a module was called an *instance* of the module, and it is assumed that a module could be evaluated multiple times (with respect to different context environments) to produce multiple instances.

Beyond this, the proposal suggests that the input environments required by a module could be produced by concatenating the environments produced as instances of other modules, assuming that the concatenated environment satisfies the module's inward specification. Thus if $C = \langle \mathit{I\text{-}spec}, \mathit{dec}\rangle$ and A and B are module instances, an instance of C could be created by the application $[\![C]\!](A; B)$, where semicolon denotes concatenation of environments. This could be expressed more explicitly by writing functions over module instances like

$$F_C \;=\; \lambda A : \mathit{O\text{-}spec}_A, B : \mathit{O\text{-}spec}_B . [\![C]\!](A; B)$$

assuming the inward specification of C is covered by the concatenation of the outward specifications O-spec_A and O-spec_B.

The final issue considered in the proposal is that types in the (inferred) outward specification of a declaration might mention type constructors inherited from its context environment, so the

outward specification and the environment itself might not be self-contained. A suggested, but rather crude, solution was to concatenate the context environment with the result environment.

Clearly this initial proposal was just a sketch of some principles that might govern the construction of environments resulting from evaluation of declarations: a kind of environment calculus. No syntactic support for defining and using modules was proposed and there was not even a way of naming modules. The main motivating objective was to support separate compilation of modules, based on inward specifications. At this point, the only mechanism to combine parts of a program into a whole was the "`use`" primitive operator for loading source files.[55]

The next major version of the modules proposal was embodied in the paper "Modules for Standard ML" by MacQueen, which was first published in the *Polymorphism* newsletter [1983b], with a revised version later appearing at the 1984 Lisp and Functional Programming Conference [MacQueen 1984]. This proposal was a detailed development of MacQueen's August 1983 design. It still viewed modules in terms of declarations and the environments they produced (still called instances) based on *antecedent* instances providing the context environment in which the declaration is evaluated. The environment types, called interfaces in the first proposal, are called *signatures* in this version. A draft of this second modules proposal was available for discussion at the June 1984 meeting and MacQueen gave a tutorial on the proposal at the beginning of the meeting.

The following new ingredients were added in this second module proposal:

- An instance (module) can contain other instances as components, thus forming a hierarchical structure of nested instances. This feature was to ensure that instances can be "closed;" i.e., any type occurring in the instance's signature is defined within the instance. Instances were declared using the `instance` keyword.
- Signature declarations were included, which allow a name to be bound to a signature.
- Module declarations were abstracted with respect to a number of instance parameters providing the context in which to evaluate the declaration. The instance parameters have specified interfaces and, thus, all modules were parameterized; i.e., all modules are *functors* in the later module terminology.
- Instances were created by applying modules to argument instances that satisfied the signatures of the corresponding formal parameters. This was called "instantiating" a module.
- The conventional dot notation was used for accessing components of an instance.
- Module instantiation was generative for datatypes defined in the module. That is, if a module contained the definition of a datatype, then each instantiation of the module contained a new version of that datatype distinct from that type component in any other instance of the module.
- Modules could contain simple type definitions (later called type abbreviations). Such type definitions have the same meaning in all instances of the module; i.e., such type components are not generative.
- If the body of a module involved creation of mutable state (typically ref values), then each module instance has its own, private mutable state. In other words, stateful declarations were also "generative," but in a dynamic rather than static sense.
- A requirement of coherence, or sharing of inherited types between instances, could be enforced by a new form of declaration appearing in the body of a module, called a *sharing* declaration.

The proposal noted that the analogue of type checking a module instantiation would involve a process called *signature matching* that compares the signature of an actual argument instance

[55] Milner and Burstall discussed possible elaborations of the idea of specifications around the end of June, 1983. These included having a compiler produce (infer) the *outward* specification of a file containing declarations [Milner 1983b].

with the signature of the formal parameter. This process allowed for polymorphic values and constructors to be specified and for such components to be used as polymorphic identifiers in the body of the module.

For programming convenience, several derived forms or shortcuts were defined, such as:

- Direct instance definitions for eliminating the definition of a module that will have only a single instance.
- Inheritance declarations for designating certain parameters as instance components of the current module.
- Opened instances for giving shortcut access to components of an instance, in both signatures and modules.
- Views for providing linguistic support for "coercing" an instance to a restricted signature (e.g., to hide components).

Finally, a section about *Foundations* briefly discusses a connection between signatures and existential types, with reference to Martin-Löf type theory and an early version of John Mitchell and Gordon Plotkin's work on the connection between type abstraction and existential types [1988].

At the June 1984 Standard ML design meeting [MacQueen and Milner 1985], various changes to the module proposal were discussed and some were adopted. Milner had prepared a proposal for an alternate syntax for modules [Milner 1984c] a couple of weeks before the meeting. This note was discussed and several of its suggestions were adopted.

One suggestion was to replace the term (and keyword) *instance* with *structure*, perhaps suggestive of "environment structure," although also suggested by the use of the term "mathematical structures" like algebras, rings, vector spaces, etc., in mathematics (universal algebra in particular). The term *module* was retained for the generating functions (abstracted declarations) that when applied produce structures.

In the proposal draft, sharing declarations had belonged in module bodies, but following Milner's suggestion, it was agreed that it would be better to have them occur in signatures. Module parameters would be considered as a single signature with multiple parameters specified as instance components within this signature and any necessary sharing specifications among components would occur within the parameter signature.

Another change was to allow free references to types, structures, etc. in a module, with these being interpreted by reference to a global *pervasive environment*. This environment would bind common predefined types like `int`, `bool`, `list`, and their associated primitive operations and exceptions.

Correspondence and the exchange of notes, questions, and clarifications on the module system continued through the remainder of 1984 and up to the May 1985 design meeting, which was a somewhat more elaborate affair organized like a workshop.[56] Once again the first day started with a session discussing the modules proposal and the discussion continued the next morning. A major issue was what it meant to "infer" the signature of a structure, and whether such an inferred signature could actually be written down in the signature syntax. MacQueen used the term *full signature* to refer to the inferred signature, which could consist either of all the information that is expressible in the syntax (with sharing specifications) or the internal static information is generated by the compiler (which is the complete static description, but it might not be possible or convenient to express it in the signature syntax).

The next full draft of the module proposal appeared in October, 1985 in the *Polymorphism* newsletter [MacQueen 1985b] and also in the Edinburgh LFCS Techreport ECS-LFCS-86-2, accompanied by

[56] And could possibly be considered the first of the long-running series of ML Workshops that continue to this day, though it was called a *design* meeting in the invitation.

a new version of the Core language proposal [Milner 1986] and Harper's I/O proposal [1986]. This version represented the state of the modules design prior to serious work on the formal Definition, in other words, the final draft of the informal design.

But this was not the final form of the module system, since further revisions, discussed below, resulted from the process of formal definition of the whole language.

5.3 Modules and the Definition

The module system was further developed and refined through the process of formally defining it as part of the work on the Definition, during the period running from 1986 through 1989, when the Definition was sent to press. The module system was also affected by further evolution of the Core language, such as the treatment of exceptions as pseudo-constructors for the `exn` type [Appel et al. 1988]. The last six chapters of the Milner and Mads Tofte's *Commentary on Standard ML* [1991] describe how the module design was formalized in the Definition.

5.4 Module Innovations in Version 0.93 of SML/NJ

Version 0.93 of the Standard ML of New Jersey compiler [Appel and MacQueen 1991] (released in February 1993). included three significant enhancements of the module system that went beyond the design of the Definition.

- Higher-order functors. This was achieved simply by allowing functors to be components of structures. This required the addition of functor signatures for specifying functor components of structures. Syntactic support included curried functor definitions. This change was based on some theoretical work by MacQueen and Tofte [1994] and an implementation design by MacQueen and Crégut [1994].
- Definitional type specifications in signatures. This feature filled an important gap in the expressiveness of signatures and also allowed some type sharing specifications to be expressed more naturally. It anticipated Harper and Mark Lillibridge's *translucent signatures* [1994] and Xavier Leroy's *manifest types* [1994].
- An alternate `abstraction` keyword for declaring structures, where a (required) signature would be by "opaque" to type information, thus providing for a way to inplement abstract types using structures. Thus, if the signature `S` specifies a type component as `type t`, and a structure is declared as `abstraction M : S = <structure expression>`, the component `M.t` is abstract. This feature partially anticipated *opaque signature ascription* (M :> S) in the *Definition (Revised)* (see below). It was also proposed in MacQueen's 1985 module design [1985b].

5.5 Module Changes in *Definition (Revised)*

The *Definition (Revised)* involved three major changes in the design and semantics of modules.

- The elimination of structure static identities. This weakened the notion of structure sharing to mean only that corresponding type components of structures would share. In other words, structure sharing became merely an abbreviation for a set of implied type sharing constraints. The main motivation for this change was to simplify the Definition by eliminating approximately three pages of rather complicated machinery that was required to handle the original strong treatment of structure sharing. This involved some loss of expressiveness that could be partially compensated for by introducing datatypes whose only purpose was to distinguish structures.
- Type specifications could include definitions (as in SML/NJ 0.93). In addition, type definitions could be applied to existing type specifications within signatures with the "`where type`" clause.

This feature was similar to the translucent types of Harper and Lillibridge [1994] and the manifest types of Leroy [1994].

- Opaque signature ascription, using the notation "`M :> S`," where simple type specifications like "`type t`," when matched, produce a new abstract type.

Unfortunately, the definitional type specification was left somewhat incomplete because of the lack of a corresponding feature for structures, which would allow bulk definitional specs for all type components of a signature. This feature was added in a later version of SML/NJ.

The higher-order functors introduced by SML/NJ were considered too radical and untested a change, and did not make it into the *Definition (Revised)*. There has been further work on the semantics and implementation of higher-order module systems has by Claudio Russo [1998], Dreyer, Russo, and Rossberg [Rossberg 2015; Rossberg et al. 2015], and by George Kuan and MacQueen [Kuan 2010; Kuan and MacQueen 2010].

5.6 Some Problems and Anomalies in the Module System

The SML module system was a major innovation of the language design and had no well-tested analogues in other languages, thus it is not surprising that some unexpected issues arose during the design and later in the development of the Definition. Here are a couple notable examples.

(1) The issue of *type inexplicitness*, which involves "inexplicit" structures, where "anonymous" or unnameable types appear in the *inferred* signature of a structure. Here is an example of an inexplicit structure `S`:

```
functor F (X: sig type t val x:t end) = struct
    val x = X.x
  end
structure S = F(struct datatype t = A | B; val x = A end);
```

The problem in this example is that `S.x` has type `X.t`, but this type path is only in scope within the body of the functor `F`. After the definition of `S` that type *exists*, but we have no way to name it. This is an odd situation, but benign.
Here is another example not involving functors.[57]

```
structure S = struct
    local
      datatype t = A | B
    in
      val x = A
    end
  end
```

This phenomenon (types or type constructors that *exist* statically by are unnameable in a particular context) need not be a problem for implementations, but may be problematic for formal static semantics.

(2) The issue of redundancy between datatype specifications in signatures and the corresponding datatype definitions in structures. The HOPE module proposal of 1981 had a potential solution to this problem in its "`datarep list is free`" declaration form. This is mainly a matter of convenience.

(3) There are other ways in which the module system is lacking fairly obvious and potentially convenient features. One such feature is parameterized signatures (signature functions taking

[57] Note that this example can be written without modules by just using the `local` declaration form.

a structure as a parameter). Work by Norman Ramsey, Kathleen Fisher, and Paul Govereau explored an number of these design possibilities [2005].

5.7 Modules and Separate Compilation

While the SML module system provides strong linguistic support for structuring programs into components with well-defined interfaces, the language design does not address the question of *separate compilation*. Instead, the Definition just specifies a program as being a linear sequence of top-level declarations. Ideally, if a program is divided in to separate units of code, it should be possible to edit and compile individual units without having to recompile the whole program. The challenge is that compilation units may have dependencies on other units; in the case of Standard ML, these dependencies can be on both the interface and the *implementation* of other units. Implementation dependencies arise because of the translucence of SML signature matching,[58] but can also arise from compiler optimizations, such as cross-module inlining.

Early in the design of the Core language, Milner explored the idea of including a *specification directive* that would allow one to specify identifiers that are used but not defined, which in turn would allow "precompilation" (Milner's term) of code [1983c, Section 4.3]. While these directives did not make it into the final design of SML, the idea of explicitly closing a compilation unit over its dependencies was the basis for the so-called *fully functorized* approach that was desribed by Tofte in a tutorial on programming in SML [1989]. In this approach, one organizes code into fully-closed functors[59] and uses functor application to "link" the final program. Since each module is explicitly parameterized over any dependencies, it can be compiled in any order with respect to the other implementation modules in the program. Furthermore, if a change is made to one module that is not reflected as a change in its signature, then it can be recompiled without having to recompile any other module. While this approach solves the dependency problem, it does not scale well to large systems.[60] Furthermore, the approach breaks down for compilers that implement optimizations, such as functor-application specialization or cross-module inlining.

As SML started to be used for more and larger systems, the need for a better compilation story was obvious. This need led to a number of efforts to define and implement separate compilation mechanisms on top of SML. One early approach, which was collaboration between the SML/NJ implementors and the Fox project at CMU provided a form of incremental (or *cut-off*) recompilation [Appel and MacQueen 1994; Harper et al. 1994]. In this implementation, if a module was modified, its antecedents in the dependency graph would not have to be recompiled. Furthermore, when a module is recompiled, if its exported interface[61] does not change, then its dependents do not have to be recompiled. This mechanism was used by the SML/NJ system for several years, but was eventually replaced by Mathias Blume's *Compilation Manager*, which provided a much richer set of build tools [Blume and Appel 1999].

Translucent signatures give rise to one form of implementation dependency, but some compilers include much more information in the exported signature of a compiler. The ML Kit compiler [Birkedal et al. 1993] is an example of such a compiler, which includes information about memory regions in the exported interface (see Section 9.2.3 for more details). To support separate compilation for the ML Kit compiler, Martin Elsman developed an approach for statically evaluating

[58] The problem with implementation dependencies via translucent signatures was avoiding in Caml Special Light by only supporting opaque signature matching [Leroy 1994].
[59] Not counting free references to types and operations from the pervasive environment.
[60] The programmer must manage dependencies as explicit functor parameters with sharing constraints, which can require many extra lines of code per module for large systems.
[61] The exported interface includes the module's signature as well as any additional semantic information that the compiler might propagate.

module-level constructs to produce residual core-level code [1999a; 1999b]. He also used a cut-off compilation mechanism similar to that used in SML/NJ.

In addition to these extra-linguistic separate compilation mechanisms, there have been languages defined on top of SML for building libraries and applications. These lanauges include the *ML Basis* system [Cejtin et al. 2004] and SMLSC [Swasey et al. 2006a]. These languages provide mechanisms to manage separate compilation, including supporting hierarchies of compilation units, visibility control, and other features. The ML Basis system was developed for the MLton compiler, but it has been adopted by other SML implementations.

Unfortunately, although there have been many interesting solutions to the separate compilation problem, these solutions have all been tied to a specific implementation. As we discuss in Section 8.11, the lack of standardization in this aspect of the language has been an impediment to portability.

6 THE DEFINITION OF STANDARD ML

Unusual for a practical programming language, Standard ML has a *mathematically rigorous definition* that is both amenable to analysis and suitable as a guide to implementation. This formal definition was, from the beginning, one of the primary goals of the language design project.

The Definition of Standard ML (or "*Definition*" for short) was developed through a series of preliminary drafts from the summer of 1985 to the end of 1989, with the final version being published by MIT Press in 1990 [Milner et al. 1990]. The Definition was revised over the period 1995–1997 to correct mistakes, to simplify both the language and its formalization, and to improve the capabilities of the module system, and was also published by MIT Press [Milner et al. 1997].[62]

6.1 Early Work on Language Formalization

Robin Milner once posed the question:

> What does it mean for a programming language to exist?

The traditional answer to this question is that there is an implementation of the language and that the language is defined by the behavior of this implementation. There may also be a language manual or book that describes the language from the point of view of programmers, but its descriptions using prose and example code will typically be incomplete and ambiguous in places. In the case of commercially successful or important languages, there may be an after-the-fact standardization process that produces a large, but not formally precise, description of what the language is supposed to be, abstracted from one or more existing implementations. A major theme in the British programming research culture going back to Christopher Strachey was that this was not good enough, and that one should strive to achieve mathematical rigor in our understanding of what a programming language is and how programs written in them behave. Thus an outstanding characteristic of the Standard ML language design was that it presupposed that the language would have a rigorous and complete formal definition and that the creation of this definition would be an integral part of the design process.

In the early 1960s, a research community developed, mainly in Europe, that was concerned with providing precise, formal definitions of programming languages. Early actors in this community included John McCarthy (Lisp, ALGOL 60), Strachey (CPL), Peter Landin (ISWIM), and members of the IFIP Working Group 2.1 (ALGOL) and the IBM Laboratory Vienna. The community organized the inaugural conference in the field, on *Formal Language Description Languages*, which was held in September 1964 [Steel, Jr. 1966].[63]

[62] The names SML'90 and SML'97 are sometimes used to distinguish between these two versions of the language definition.
[63] See Astarte's Ph.D. dissertation for further details [2019, Chapter 4].

McCarthy had already provided a definition of Lisp in the form of a definitional interpreter [1960]. The IBM Laboratory Vienna, led by Heinz Zemanek, developed a definitional interpreter for PL/I over the period 1965-1967, using formalisms that became the Vienna Definition Language (VDL) [Lucas et al. 1968]. Strachey was developing concepts for language description in the mid 1960s [1967] that would evolve into the denotational semantics of Dana Scott and Strachey [1971] toward the end of the decade. Landin used the lambda calculus as a tool for describing ALGOL 60 [1965a; 1965b]. Adriaan van Wijngaarden and colleagues developed a formal description of ALGOL 68 [Lindsey 1996; Mailloux et al. 1969; van Wijngaarden et al. 1969] using two-level grammars.

The development of the PL/I definition by Zemanek's group at the IBM Laboratory Vienna began after the original informal design by a committee consisting of IBM engineers and customers (from the SHARE user group) was complete. Zemanek's group took a definitional approach that was influenced by earlier work by McCarthy [1963] and Landin [1964]. This effort resulted in the Vienna Definition Language (VDL) [Lucas et al. 1968] description of PL/I, which was basically a definitional interpreter focused on the dynamic semantics of the language.[64]

The ALGOL 68 design came with a partial formal description using enhanced syntactic techniques (two-level grammars), but this formalism did not cover dynamic semantics and it provided a baroque and incomplete specification of the type system and other context-sensitive aspects of the language. There was a strong reaction to the complexity of this definition, leading to a breakup of the IFIP Working Group 2.1 that was responsible for ongoing development of the ALGOL language [Astarte 2019, Section 7.1].

The Standard ML definition differs from these earlier efforts in several respects.

- The formal definition was a first-order goal of the original designers of the language and (a subset of) those designers developed the formal definition themselves.
- The formal definition was intended to support reasoning about the metatheory of the language (e.g., type soundness and the determinacy of the dynamic semantics).
- By the early 1980s, more mature and suitable formal definition techniques were available that were not available to earlier designers, such as Structural Operational Semantics [Plotkin 1981, 2004] and *Natural Semantics* [Kahn 1987].

One problem with designing a language and then later formalizing it is that the informal design may explicitly or implicitly involve features that are difficult to describe formally (this obviously depends in part on the choice of the formalism chosen), leading to either failure of the formalization, incomplete definition, or excessive complexity in the formal definition.[65]

In the case of Standard ML, the *design* of the language was not considered finished until the *definition* was completed. A much tougher criterion for being finished would be to have certain specified metatheory theorems *proved*. In effect, there are two levels of design: the language itself and the formal definition and its formalisms and techniques. Proving metatheoretical results for a real language is also a major challenge, because of scale.

Ideally, two goals ought to be satisfied before the full completion of a language design:

(1) the development of a formal definition with metatheory, and
(2) at least one complete, debugged, tested implementation.

Serious problems in the first goal might require revising the language and/or the formal definition, while serious problems in the second require revision of the language design, which would force revision of the formal definition. A number of programming language designs, such as PL/I, ALGOL 68, and Ada, proved very challenging to implement given the state of compiler technology

[64] Astarte's Ph.D. dissertation gives a detailed recounting of the history of the PL/I definition [2019, Chapter 5].
[65] Conversely, tailoring a language design to make formal definition more convenient might lead to awkward or unnatural design choices, which is a danger to be taken into account in choosing the formalism.

at the time, which resulted in long delays before adequate implementations became available. In the case of Standard ML, several implementations were developed during the design period of 1983 to 1989 — including Edinburgh SML [Mitchell and Mycroft 1985], Poly/ML [Matthews 1989], and Standard ML of New Jersey [Appel and MacQueen 1991] — during the design period of 1983 to 1989, and experience with these implementations had some influence on the design and definition of the language.

6.2 Overview of the Definition

The Definition defines the following aspects of the language:

(1) Its *concrete syntax*, given by a context-free grammar that defines the syntactic categories of the language as sets of strings, from which the *abstract syntax* may be inferred.
(2) Its *static semantics*, or *statics*, which defines context-sensitive constraints on programs, such as typing and variable scoping, that guarantee that programs are well-formed and sensible.
(3) Its *dynamic semantics*, or *dynamics*, which defines how to execute a program relative to a dynamic environment and a *store* of mutable data structures and exception constructors.

The definition of the grammar of the language, given in Sections 2 and 3 of the Definition, follows the practice used in LCF/ML [Gordon et al. 1979, Section 2.2]. The concrete syntax is described using a BNF-like notation extended with a "..." notation for repetitive sequences, "L" and "R" annotations to indicate left or right associativity, and a convention that the precedence of constructs is determined by the order of the productions. The latter property is specified by the statement

> The constructs are listed in order of decreasing binding power.

But the meaning of "binding power" is left undefined. There is also a general rule that the extent of iterated constructs should run as far to the right as possible.[66] Because of the annotations and the precedence convention, the grammar can dispense with phrase classes (nonterminals) used only to deal with issues of precedence, associativity, and iteration, leaving only phrase classes that express the logical components of programs (e.g., expressions, declarations, types), so it is possible to regard the grammar as also serving as the *abstract syntax* of the language, which is how it is used in the inference rules making up the core of the Definition.

The language was factored into a Core language and a Module language, each with its own static and dynamic semantics, with a relatively clean interaction between the semantics of the two levels. The Core language was further divided into an essential, minimal set of constructs (the *Bare* language) from which the rest of the Core constructs could be derived. For instance, the conditional-expression form

```
if e1 then e2 else e3
```

was defined to be syntactic sugar for the case expression

```
case e1 of true => e2 | false => e3
```

which in turn is modeled by an application of a clausal function to an argument:

```
(fn true => e2 | false => e3) e1
```

Hence the semantics only had to deal with constructs of the Bare language.

[66] This rule means roughly that shift/reduce conflicts should be resolved in favor of shift.

6.3 Early Work on the Semantics

While Michael Gordon and Milner had written, but not published, a denotational semantics for LCF/ML in the late 1970s [Milner et al. 1990, Appendix E], that approach was abandoned for the definition of Standard ML — instead, an operational approach was adopted. While the reason for this choice is lost to the sands of time, it is likely because of the practice of using operational techniques that had been growing in Edinburgh in the late 1970s and early 1980s as exemplified by Gordon Plotkin's famous notes on *Structural Operational Semantics* (SOS) [1981; 2004]. These operational approaches had several advantages over denotational techniques.

- They can handle concurrency more simply and elegantly.
- They can be used to express both dynamic *and* static semantics of a language (as in the case of Standard ML).
- They are amenable to proving metatheoretic results such as type soundness.

The eventual semantics of SML (both static and dynamic) were expressed using the style of inference rules described in Section 4.1. This choice appears to have been influenced by the use of evaluation semantics in Per Martin-Löf's influential paper [1982] and by Gilles Kahn's Natural Semantics [1987]. Recall that these rules are used to prove judgments of the general form

$$C \vdash P \Rightarrow M$$

where C is a context (usually an environment-like structure), P is a phrase of the language, and M is the semantic representation of the meaning of P in the context B. For static semantics, the meaning is a *semantic object*, which might be a type, in the case where P is an expression, or a much more complicated construct in the case where P is a functor, for instance. In any case, the semantic object is intended to capture the static information produced by analysis of the phrase. The process of translating a phrase to a semantic object representing its static content is called *elaboration*. In a dynamic semantics, the translation produces a representation of "values" or dynamic "environments" that result from the evaluation of the phrase. In practice, a natural semantics for dynamics (also known as an *evaluation semantics*) is usually fairly similar to an interpreter or can easily be reformulated as interpreter.

The development of the semantic description of Standard ML began in the spring of 1985 (if not earlier) with a proposed dynamic semantics written by Milner [Milner 1985a].[67] That summer, after Robert Harper arrived in Edinburgh and the third design meeting was held in May, Harper began work on the first version of a static semantics, focusing on the module system, since type checking for the Core was assumed to be mostly covered by Milner's previous work on polymorphic type inference.[68]

In response to Harper's draft, Milner produced an alternate semantic model that he called "Webs" [Milner 1985c]. The abstract and minimal Webs model was narrowly focused on describing the effect of structure-sharing specifications in terms of trees of paths with congruence relations. A *web* is a pair (T, C), where T is a *tree* (a prefix-closed set of paths, where a path is a sequence of labels), and a congruence relation is an equivalence relation on paths of a tree that respects path extension (i.e., if C is a congruence relation on paths of a tree and and $C(p1, p2)$ then $C(p1.a, p2.a)$ if $p1.a$ and $p2.a$ are also paths in the tree). The tree represents the substructure hierarchy of a structure and the congruence relation represents sharing among components of the tree (i.e., substructures). Harper had used a notion of "timestamps" to uniquely identify structures for the

[67] It is uncertain when Milner began work on this, since this document from April, 1985 is labelled as the "3rd Draft."
[68] Harper also started implementing the module system that summer as part of the project to retrofit the Edinburgh ML compiler as a Standard ML compiler.

$$
\begin{array}{rcl}
m, n & \in & \text{Names} \\
M, N & \in & \text{Fin(Names)} \quad \text{i.e., finite subsets of Names} \\
E & \in & \text{Env} = \text{StrId} \xrightarrow{\text{fin}} \text{Str} \\
S \textit{ or } (n, E) & \in & \text{Str} = \text{Names} \times \text{Env} \\
\Sigma \textit{ or } (N)S & \in & \text{Sig} = \text{Fin(Names)} \times \text{Str} \\
[\textit{strid}, \Sigma, \textit{strexp}, \mathcal{E}] & \in & \text{Fun} = \text{StrId} \times \text{Sig} \times \text{StrExp} \times \text{ProgEnv} \\
F & \in & \text{FunEnv} = \text{FunId} \xrightarrow{\text{fin}} \text{Fun} \\
G & \in & \text{SigEnv} = \text{SigId} \xrightarrow{\text{fin}} \text{Sig} \\
\mathcal{E} \textit{ or } (M, F, G, E) & \in & \text{ProgEnv} = \text{Fin(Names)} \times \text{FinEnv} \times \text{SignEnv} \times \text{Env}
\end{array}
$$

Fig. 1. Semantic Objects for ModL [Harper et al. 1987b, Table 1], where StrId, FunId, and SigId are (resp.) structure, functor, and signature identifiers, and Names is a set of names used to uniquely identify structures.

purpose of sharing,[69] and Milner's Web model avoided this device by representing sharing directly as congruences. But congruences were not a very practical representation, and modeling functors was rather complex. Also, there were no types in the minimal language, so the issue of how sharing would interact with type checking was not addressed.

The next step towards a static semantics for modules was the paper "A type discipline for program modules" [Harper et al. 1987b], which was presented at the TAPSOFT conference in March 1987. This paper gave a static semantics for a simplified module language, called *ModL*, with structures, functors, and signatures with sharing, but no types or expressions. The paper followed Harper's original idea of representing sharing via timestamps (or unique static identifiers), which the paper called "names."[70]

Table 1 of the paper defined the semantic objects for the model; we reproduce it here in Figure 1. The most interesting objects are the environments, structures, signatures, and functors. A structure S is represented by a name n that statically identifies that structure for the purpose of sharing, paired with an environment E that gives the components of the structure. A signature of the form $(N)S$ consists of a structure, some of whose names, N, (i.e., the name of the structure itself and of some of its component substructures) are *bound*, in the sense that they can be instantiated when matching the signature with a compatible structure. A functor object of the form $[\textit{strid}, \Sigma, \textit{strexp}, \mathcal{E}]$ represents a closure (with respect to a top-level or program environment $\mathcal{E}$) of a lambda abstraction over *strid* with type Σ and body *strexp*. The closure environment binds any free structure or functor identifiers contained in the body *strexp*.

Figure 2 of the TAPSOFT paper gives the inference rules that define the elaboration of structure-level declarations to environments and structure expressions to structure, and Figure 3 gives rules for elaborating (signature component) specifications to environments, signature expressions to signature objects, and programs to program environments. We survey a few of the rules from Figure 3 here; the interested reader is directed to the paper for more details.

When elaborating a basic structure expression to produce a structure object, we can freely choose any name to identify it, with the constraint that it is a "fresh" name not previously used (i.e., not in the program environment's "reserved" name set). The interesting aspect is how names are chosen

[69] Harper's draft semantics has been lost, but we know that it used timestamps because of a comment Milner makes in the Webs paper.

[70] The "name" terminology, meaning internal static tokens used for determining static equality or sharing, is rather unfortunate, given the many possible meanings for the term in programming language theory and implementation. In SML/NJ for instance, the terms "stamp" or "uid" are used for this concept.

when elaborating signatures. The names associated with the structure part of a signature object are completely arbitrary in the rules

$$\frac{\mathcal{E} \vdash spec \implies E_1}{\mathcal{E} \vdash \texttt{sig}\ spec\ \texttt{end} \implies (\emptyset)(n, E_1)} \quad \text{[Harper et al. 1987b, Rule 9]}$$

and the set of bound names is determined arbitrarily after the basic signature structure.

$$\frac{M, F, G, E \vdash sigexp \implies (N)S \qquad n \notin M}{M, F, G, E \vdash sigexp \implies (N \cup \{n\})S} \quad \text{[Harper et al. 1987b, Rule 11]}$$

But for those names to work in the context of a larger elaboration, they must have been chosen to satisfy all sharing equations. Furthermore, the set of bound names has to have been chosen to produce a *principal* signature, which means that all the names that are not constrained by sharing to be identical to a fixed name in the context should be included in the bound set.

$$\frac{\mathcal{E} \vdash sigexp \implies \Sigma \qquad \Sigma \text{ principal for } sigexp \text{ in } \mathcal{E}}{\mathcal{E} \vdash \texttt{signature}\ sigid = sigexp \implies (\emptyset, \emptyset, \{sigid \mapsto \Sigma\}, \emptyset)} \quad \text{[Harper et al. 1987b, Rule 14]}$$

In effect, there appears to be a lot of freedom in choosing names in signatures and choosing which names should be bound, but some choices will not "work," that is, will not allow a valid derivation from the rules to be constructed, so we nondeterministically use some set of choices that happen to "work," and there will in general be an infinite number of these successful choices. An elaboration *algorithm*, on the other hand, is likely to use a strategy to deterministically make the right choices (as in the case of polymorphic type inference, where the typing rules require correct "guesses" to make a derivation work).

The ModL language studied in the TAPSOFT 87 paper is degenerate because of the lack of types and expressions means that there is no type checking, and the whole purpose of sharing equations is to make sure that type components of structures will match during type checking. Nevertheless, the approach served as a test run for the more complete static semantics of the whole language in the Definition.

6.4 The Definition

Building on the TAPSOFT 87 paper, the completion of the Definition required adding value-level types, expressions, and declarations. The starting point was the description of Standard ML given by the Edinburgh technical report ECS-LFCS-82-6 [Harper et al. 1986], which contained three language proposals: "The Standard ML Core Language (Revised)," which was Milner's final draft of the Core language description; "Standard ML Input/Output" by Harper, dating from June 1985; and "Modules for Standard ML" by David MacQueen, dating from October 1985. Before the first version of the Definition was produced, the Core language design was amended, to include the following changes [Milner 1987]:

- the notation for numeric record labels was changed and their use was liberalized,
- record selector functions were introduced,
- optional `withtype` clauses were added to datatype declarations,
- equality type variables were introduced and the class of equality types that could instantiate them was defined, and
- the polymorphic assignment operator "`:=`" was added, with the technical restriction that it could only be instantiated to monotypes in expressions.

With these changes, the first version of the Definition, titled "The Semantics of Standard ML (Version 1)" (later versions were titled "The Definition of Standard ML") was published in August of 1987 as an Edinburgh technical report [Harper et al. 1987a].

The second version appeared in August 1988 [Harper et al. 1988]. In addition to technical corrections, the notable changes made in this version were

- a bibliography and index were added,
- exceptions were made constructors [Appel et al. 1988],
- Section 8 on Programs and the interactive top level was added,
- derived forms for functor definition and application (structures as parameters) were added,
- a section on "Signature Matching" was added,
- imperative type variables were added to allow polymorphic references, and
- restrictions were placed on the typing of polymorphic references and exceptions.

A new historical appendix: "The Development of ML" was was also added in Version 2.

The third version appeared as a technical report in May 1989 [Harper et al. 1989]. This version made relatively few changes relative to Version 2, with the most significant being that that signatures must be *type explicit*, which means that any occurrence in a signature of a type specified in the signature must be in the scope of that specification, which basically prevents pathological signature definitions that contain multiple specifications of a type identifier. An addendum was added to the beginning of this tech report in January 1990 specifying differences between it and the published version of the Definition. The changes introduced in the published version are mostly minor and technical, except for a major revision of the description of *principal signatures* and *equality principal signatures* are defined [Harper et al. 1989, Section 5.13].[71]

The version of the Definition published by MIT Press [Milner et al. 1990] consists of the following chapters and appendices:

1 Introduction (2 pages)
2 Syntax of the Core (7 pages)
3 Syntax of Modules (6 pages)
4 Static Semantics for the Core (15 pages, 52 rules)
5 Static Semantics for Modules (15 pages, 50 rules)
6 Dynamic Semantics for the Core (11 pages, 56 rules)
7 Dynamic Semantics for Modules (6 pages, 36 rules)
8 Programs (3 pages, 3 rules)
A Derived Forms
B Full Grammar
C The Initial Static Basis
D The Initial Dynamic Basis
E The Development of ML

This structure illustrates the factoring of the definition between the Core language (actually the minimal Bare language) and the Modules language. For each of these languages there is a division between static and dynamic semantics. The Definition did not give a complete description of the syntax of the language, but rather spread the syntax across multiple chapters and appendices. Lexical issues were addressed in Chapter 2, and the bare syntax of the language was covered in Chapters 2 and 3. The syntax of the Bare language was extended to the full language via a collection of derived forms that were defined in Appendix A. Lastly, the full grammar for the Core (but not modules) was given in Appendix B. In addition to being spread out over multiple chapters, the

[71] These concepts are explained in detail in the Commentary on Standard ML [Milner and Tofte 1991, Chapter 11].

grammar contained a number of (benign) ambiguities. It is ironic that the focus on using the syntax as an abstract syntax for defining the language's semantics led to the less rigorous description of the concrete syntax. Appendices C and D define the *Initial Basis*, a relatively sparse environment of predefined and pervasively available types, values, and functions (see Section 8 for more details).

At the heart of the semantics are the inference rules. These rules are fairly evenly divided among the four categories: static Core, static Modules, dynamic Core, and dynamic Modules. In the remainder of this section, we take a closer look at some of the technical aspects of the Definition, which the casual reader may wish to skip

As shown in Figure 1, the ModL language from the TAPSOFT 87 paper had only nine forms of semantic objects. In the case of the Definition, with the addition of types, values, and exceptions, there were now 27 forms of semantic objects for just the static semantics of the Core [Milner et al. 1990, Figure 10, Section 4.2] and Modules [Figure 11, Section 5.1]. For instance, the representation of types required the following semantic objects:

$$\begin{array}{rcl} \theta \text{ or } \Lambda\alpha^{(k)}\tau & \in & \text{TypeFun} = \bigcup_{k\geq 0} \text{Tyvar}^k \times \text{Type} \\ (\theta, CE) & \in & \text{TyStr} = \text{TypeFcn} \times \text{ConEnv} \end{array}$$

where τ ranges over (representations of) type expressions, and CE ranges over *constructor environments*, which map Con (data constructor) identifiers to their types. A TypeFun object represents a simple function over types defined by lambda-abstracting a finite sequence of type variables over a type expression. Thus a TyStr can model a type constructor defined by a simple type declaration such as "`type ('a,'b) t = 'a * 'b`," with a representation of the form $(\theta, \emptyset)$, where θ is roughly $\Lambda(\alpha, \beta).(\alpha \times \beta)$ and an empty constructor environment. On the other hand, a datatype d would be represented as a TyStr of the form $(\Lambda().n, CE_d)$ where n is a unique name identifying the datatype and CE_d specifies the types of the data constructors for d. Thus the two forms of type constructors (simple and datatype) use the same TyStr representation, but each is degenerate in its own way and there are no type constructors that have both a nontrivial θ component *and* a nonempty CE.[72]

Semantic objects for structures and signatures are similar to the corresponding objects for ModL. A structure object (Str) is a pair of a name (unique identifier) and an environment, such as (n, E), but in this case the environment part is a compound 4-tuple of environments, $E = (SE, TE, VE, EE)$ where SE is a structure environment mapping structure variables to structure objects, TE is a type constructor environment mapping type constructor identifiers to TyStr objects, VE is a variable environment mapping a set of variables, data constructor identifiers, and exception constructor identifiers to their, possibly polymorphic, types (represented as TypeScheme objects), and finally EE is an environment mapping exception constructor identifiers to their type. We need these four kinds of environment to represent the four kinds of components that can be contained in a structure in Standard ML.

A signature object is now a Str together with a finite set of bound names, but now names come in two flavors: TyName and StrName, so there are actually two name sets:

$$\begin{array}{rcl} N \text{ or } (M, T) & \in & \text{NameSet} = \text{StrNameSet} \times \text{TyNameSet} \\ \Sigma \text{ or } (N)S & \in & \text{Sig} = \text{NameSet} \times \text{Str} \end{array}$$

As in ModL, the NameSet identifies components of the body structure that are bound and available to be instantiated in the process of matching a structure against the signature. The Commentary uses the term *flexible* for bound names (and *rigid* for free names).

It turns out that there are a number of ways that structures or signatures can be ill-formed, so there are a number of attributes that distinguish the good module objects:

[72] If this seems a bit baroque, no one could argue the point!

- consistency (Section 5.2): corresponding (static) components of structures that share also share (have the same "name" or are equal),
- well-formedness (Section 5.3): components of a rigid structure (having a free name) are also rigid,
- cycle-freedom (Section 5.4): the name of a structure does not occur as the name of a component,
- admissible (Section 5.5): a structure is admissible if it is consistent, well-formed, and cycle-free, and

It is the case that in any successful derivation of a judgement from the inference rules, all the semantic objects occuring in the derivation will be admissible.

When deriving the semantic representation of the structure part of a *sigexp* in a given context, the inference rules do not determine which names should be bound. Therefore, the Definition has a special inference rule that requires the bound names to be chosen to make the signature principal [Milner et al. 1990, Rule 65]; i.e., to bind as many names as possible, which means any names that are not rigid because of sharing with rigid structures.[73]

The inference rules for the static semantics of modules make use of several relations and mechanisms to model features such as signature ascription and functor application.

- *Realization*: roughly a mapping from bound names to rigid objects, which, when applied to the Str part of a Sig, produces a rigid Str of the same "shape" as the signature (same component names).
- *Signature instantiation*: $\Sigma_1 \geq S_2$. There is a realization ϕ that when applied to S_1 (where $\Sigma_1 = (N_1)S_1$) yields S_2; i.e., $\phi(S_1) = S_2$, and the domain of ϕ is a subset of N_1.
- *Functor signature instantiation*: a similar relation for functors and functor signatures.
- *Enrichment*: $S_1 \succ S_2$. S_1 has at least as many components as S_2 and where their components agree, S_1 has more complete information (and similarly for environments and type constructors).

Given these mechanisms and relations, the notion of signature matching is defined, which specifies when a structure is compatible with a signature (i.e., when does it satisfy the interface specification given by the signature). A structure S matches a signature Σ if there exists a structure S' such that $\Sigma \geq S' \prec S$. This means that there is a realization of Σ that can be enriched to produce S. The realization replaces bound names to match those of S and then extra "stuff" can be added to get to S. The result of the signature match is the intermediate structure S', which will have the same "shape" as Σ but the same "names" as S.

Signature matching is used in the inference rules to model what happens when a signature is ascribed to a structure and what happens when a functor is applied to an actual argument.

This discussion is meant to give a rough impression of how the static semantics models the module system. There are many details worked out through the definition of other semantic objects (such as functors, which we have not mentioned), and there are 196 inference rules that specify the details of the modeling. For a much more detailed explanation of the semantics, see the Commentary [Milner and Tofte 1991].

The dynamic semantics follows the same general pattern, except that phrases are transformed into objects modeling the values and environments that result from evaluation of a phrase or program.

[73] In fact, the requirement is a bit stronger, the bound names must make the signature object *equality principal*.

6.5 The Revised Definition

For Milner, the publication of the Definition was the end goal of the Standard ML project, and he had other topics competing for his attention, mainly his research on the semantics of concurrency, including continuing work on process algebras such and his "Calculus of Communicating Systems" [Milner 1980] and its successors.

But implementations of the language had begun to be available as early as 1985, even before the design was complete. By the late 1980s there were at least four full or partial implementations (Edinburgh Standard ML, ML under Unix, Poly/ML, and Standard ML of New Jersey) and a growing user community. These implementations, plus the availability of early textbooks on the language, led to a growing user community, which had an interest in the further development of the language and its implementations.

There were also new language features and capabilities that were being introduced by particular implementations, as discussed in Section 5.4, or studied by researchers such as Harper and Mark Lillibridge [1994] and Xavier Leroy [Leroy 1994]. Critiques, such as Stefan Kahrs's [Kahrs 1993], pointed out problems with the Definition. In addition, in the background, the ML2000 group had begun meeting in the early 1990s to explore future evolution of the language or successor designs. All of these activities created pressure to revisit the design and definition of Standard ML.

The Newton Institute program on the Semantics of Computation took place in Cambridge in the second half of 1995 and it brought together the four people most involved in the design and definition of the language, namely Milner, Harper, Mads Tofte and MacQueen (both MacQueen and Tofte were also involved in implementing SML compilers). It was jointly deemed to be a good time and opportunity to re-open the issue of revising the Definition. Most of the discussions about possible changes took place in Cambridge that autumn, with discussion continuing by email over the next year or so. These discussions eventually led to the publication of "The Definition of Standard ML (Revised)" in 1997 [Milner et al. 1997].

The major changes in the revised language involved the type and module systems [Milner et al. 1997, Appendix G]. For the type system, the most important change was the replacement of "imperative polymorphism" by the "value restriction" as discussed previously in Section 4.4. Another change was the introduction of datatype replication declarations and specifications, which fixed a clear flaw in expressiveness.

In the module system, there were three major changes.

- Type abbreviations in signatures: this change endorsed a modification made in SML/NJ 0.93 and explored theoretically by the Harper and Lillibridge [1994] and Leroy [1994], but with a new "`where type`" clause for externally specifying such definitional specifications on existing signatures.
- Opaque signature matching: this feature complemented the above addition to give finer control of visibility and abstraction via the module system, making `abstype` essentially redundant.
- Downgrading of structure sharing: this change weakened the meaning of structure sharing, which became an indirect form of type sharing.

These changes have been discussed above in Section 5.5. The change in structure sharing was motivated mainly by the simplification it would produce in the semantics of modules. The total number of inference rules in the Definition (Revised) was reduced from 196 to 189.

There were other changes that were discussed but were not in the end adopted. The most controversial was probably the idea of eliminating equality type variables and equality polymorphism. Having already dismissed imperative type variables, this would have left only a single simple form of type variable, which would have been a significant clean-up of the type system. On the practical

side, no matter which implementation strategy is used, the problems with polymorphic equality (discussed in Section 4.5) make it a poor tool in the general case, so its use tends to be limited.

One change that was omitted from the revision was definitional specifications for structures and a corresponding "**where structure**" clause for signature expressions. The omission of this feature meant that there was a proliferation of type-level definitional specifications, in some cases, when converting code from the old structure sharing style to use the new features. Some implementations (including SML/NJ) added this feature as a non-standard extension, because of its usefulness.

One external factor that inhibited some changes during the deliberations was that Harlequin, a local software house, had started a project to build a commercial Standard ML compiler, eventually released as Harlequin ML Works [Haines 1991]. Although the Standard ML designers had good relations with the developers at Harlequin, the president of the company, Joe Marx, wrote an indignant letter to Milner objecting to the prospect of a change in its definition that might affect his company's product plans.

7 TYPE THEORY AND NEW DEFINITIONS OF STANDARD ML

Once the Definition was completed at the end of 1989, was the language finished? Was there any point in further research involving Standard ML? The obvious answer was that Standard ML had become a living language, because of the Definition, the availability of implementations, and the existence of an active user community. But there was also an active research community interested in investigating questions arising from the design and the Definition, and one of those questions was whether the language or its definition could be improved. Already in the mid-1980s, just as work on the Definition was getting started, there were new developments connecting type systems or typed calculi to the foundations of the language, and in particular the static semantics of modules [MacQueen 1985a]. Over the next twenty years or more, a rich research literature on the connections between Standard ML and type theory developed. This section surveys some of important accomplishments of that period, including a complete mechanized semantics in the Twelf theorem prover.

The biggest challenge was to develop a type theory for program modules. Two major themes influenced the development: *abstract types* and *modular structure*. The two are not independent and their relation involves the interplay between *opacity* and *transparency* of type definitions. One way of achieving data abstraction is based on John Reynolds' polymorphic typed λ-calculus [1974], which was independently discovered by Jean-Yves Girard [1972] in the context of formal logic.

These ideas were extended by John Mitchell and Gordon Plotkin in their influential 1985 paper on modeling abstract types as existential types [1988]. In their scheme, an expression of an existential type $\exists t.\tau$, interpreted as an abstraction, is a *package* of the form $abst = \langle \tau', body \rangle$ consisting of a *witness* τ' for the existentially quantified type variable t and an expression *body*, with type τ, that normally implements an interface of operations for t. To use the abstraction in an expression *client*, it must be *opened* using a special **let**-like construct, "**open** $(t, x) = abst$ **in** *client*." Abstraction is enforced by the fact that the identity of the type t is not known in the expression *client* and the name t is not in scope outside the **open** expression, nor may it occur in the type of that expression. However, for practical uses in modular programming, abstract types as existential types had a significant flaw: the lowest level modules would have to be opened with the widest scope to be available to all of their clients. The hierarchy of scopes would therefore be the inverse of the hierarchy of module dependencies.

This problem was pointed out by David MacQueen, who developed a type-theoretical foundation for modules [1986] partly inspired by Per Martin-Löf's work on intuitionistic type theory [1982; 1984]. This type-theoretic semantics used general sums (Σ) and general products (Π), also sometimes called *dependent types*, to express hierarchy and parameterization in program structure.

MacQueen's analysis introduced *type sharing specifications* to express coherence constraints among the arguments to a parameterized module (as explained in Section 5.1).

MacQueen's dependent types avoid the need for an **open** construct by providing direct access to the type components of a module, using the familiar "dot notation" (e.g., `M.t`). But MacQueen's approach raises a question: since modules contain both type definitions and definitions of values that have to be dynamically computed, can the type components of a module `M` depend on dynamic execution, which may produce *effects* like divergence, mutation of state, and the raising of exceptions? Harper, Mitchell, and Eugenio Moggi introduced the idea of the *phase distinction* [1990; 1991], which establishes the conditions under which the static semantics can be elaborated independently of the dynamic semantics, thus ensuring that static properties like types are not affected by dynamic execution at the value level. It turned out that these conditions were not difficult to satisfy and in fact MacQueen's module language did satisfy them.[74]

The typed calculus model presented by MacQueen was further refined by Robert Harper and Mitchell in the paper "The Essence of ML"[1988] and in an extended version "On the Type Structure of Standard ML" [1993]. They model the static semantics of both the Core and Module language of Standard ML using a typed calculus *XML*, which is based on a predicative version of the Reynolds-Girard polymorphic lambda calculus (F_ω). They considered whether module sharing equations could be modeled by Martin-Löf' equality types [Martin-Löf 1982], but discovered technical problems with that approach [Harper and Mitchell 1993, Section 9.3].

A problem with MacQueen's original design of the module system and the type calculi for modeling its static semantics is that there is no support for type abstraction. The default is that a type exported from a module (even as mediated by a signature ascription) is fully known (or "concrete"). The module language provides no mechanism to export a type as abstract, where its actual definition is hidden. The Core language provides **`abstype`** declarations for creating new abstract types, but it was felt that the module language should also be able to support abstraction as well. There was also the fact that a type specified as "**`type`** `t`" in a functor parameter signature was effectively abstract with respect to the body of the functor definition, simply because its instantiations by potential arguments of the functor were unknown.

The first type theories to add support for abstraction to the module language were developed in a series of papers by Xavier Leroy from 1990 to 1996 [1990; 1994; 1996], in support of the design of the OCaml module system, and by Harper and Mark Lillibridge [1994] and in Lillibridge's thesis [1997] in an effort to consolidate the existential and dependent formulations of modular structure. Practically speaking, a new variety of *opaque* signature ascription (notation: `M :> SIG`) was added that made types exported through the signature abstract by default, but this could be counteracted for individual types by using more precise, "definitional" specifications like "**`type`** `t = int list`" that included a definition of the type. Leroy called types exported in this way *manifest* types, while Harper and Lillibridge used the term *translucent types*. These proposals extended MacQueen's 1985 fully opaque "abstraction" declaration to admit type definitions, which MacQueen had, independently of these theories, developed and implemented in the February 1993 release of SML/NJ (Release 0.93).

The type systems developed by Harper and Lillibridge and by Leroy rely on a delicate interplay between dot notation, subtyping, and type ascription to enforce abstraction. These mechanisms were studied by Aspinall [1995] and by Harper and Chris Stone [2000; 2000; 2006], who developed the theory of *dependent singleton kinds* to model type abstraction and type definitions in module systems. For a type to have a singleton kind means that it is equal to the type specified by the singleton (think of a singleton set, which has one element). Thus, in particular, if a type identifier

[74] One simplifying factor was that the module system only allowed type projections like `M.t` on module identifiers, not allowing projection of types from arbitrary module expressions like `F(M).t`.

is declared with such a kind, it amounts to a type definition. Type-sharing relationships can be expressed by choice of representative of each equivalence class, assigning singletons of that representative to each of the other elements of the class. Singleton kinds are a form of dependent type, albeit one that does not violate the phase distinction. Correspondingly, product and function kinds are generalized to dependent forms.

In these type systems for modules the use of "dot notation" to access the type components of a module is limited to a distinguished class of *module values*, which include paths, or iterated projection from variables, as in Standard ML. This ensures that type equality is well-defined and decidable, but, more importantly, it is an expression of the notion of "type generativity" in the language. In particular any module computation that might incur an effect, such as allocation of a reference cell, must be bound to a variable before its components can be accessed. Because bound variable names are subject to implicit renaming to avoid collisions with others, the type components of a bound module are "fresh" in the sense of being distinct from all other types in scope. This idea was further developed by Derek Dreyer, Karl Crary, and Harper [2005; 2003], which accounted for "weak sealing" as it is found in OCaml, and by Daniel Lee, Crary, and Harper [2007], which was used in the mechanization of the Standard ML definition described below.

7.1 Reformulating the Definition Using Types

In light of these developments it was natural to reconsider the methods used in the Definition to define the static semantics of Standard ML. This idea was taken up in the late 1990's by Claudio Russo [1999a; 1999b; 1998] and by Martin Elsman [1998; 1999a] using similar methods. In particular they were both concerned to clarify the concept of "type generativity" used in the Definition, and to consider extensions to the language to support separate compilation, and first-class and higher-order modules.

A central point of their work is that type-name binding in the Definition is used to express three separable concepts, parameterization and universal and existential quantification. When properly distinguished, these concepts may be used to clarify the static semantics. For example, in their reformulations the static semantics of a functor has the form $\forall T.(\Sigma \longrightarrow \exists T'.\Sigma')$, in which T is a *provided* set of type names, Σ is the signature of the argument structure, T' is the *generated* set of type names, and Σ' is the signature of the result. The names in T may occur in both the argument and result, expressing a form of type dependency, but the names in T' may only occur in the result, being "new" on each application of the functor. Russo proved that the existential quantifier accurately models type generativity as formulated in the Definition, justifying the reformulation in these terms.

With this framework Russo and Elsman studied several extensions to Standard ML, including first-class modules, which are closely related to Mitchell and Plotkin's existential types, higher-order modules, which permit functors to take and return other functors, and, most importantly, separate compilation of program units. As practical validation, Russo's reformulation was used in the Moscow ML compiler [Moscow-ML 2014] and Elsman's was similarly used in the MLKit compiler [Birkedal et al. 1993].

Because they dealt only with the static semantics of modules, these reformulations are not suitable for proving metatheoretic properties such as type safety for the language. Later, Andreas Rossberg, Russo, and Dreyer [2015] extended the statics with a translation into F_ω to account for the dynamic semantics as well. Their approach follows the reformulation described in the next section, but, rather than providing a type theory for modules, they instead incorporate a translation based on the phase distinction into the elaboration process.

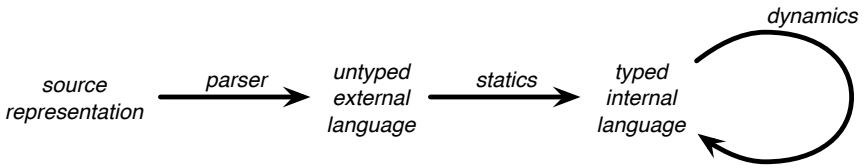

Fig. 2. Structure of the Type-Theoretic Formalization of Standard ML

7.2 A Mechanized Definition

Shortly after Russo and Elsman's reformulations of the Definition were published, Harper and Stone [2000] proposed a type-theoretic interpretation of Standard ML that accounted for both the statics and dynamics of the language to support both implementation and metatheory. The first application of the interpretation was to extend the TIL compiler [Morrisett 1995; Tarditi et al. 1996, 2004] to support modules [Petersen et al. 2000; Petersen 2005], and separate compilation [Swasey et al. 2006a,b].

The Harper-Stone interpretation was later refined by Lee et al. [2007] in their mechanization of the Definition using Twelf [Pfenning and Schürmann 1999]. The overall structure of the interpretation is illustrated in Figure 2. The statics takes the form of an elaboration of Standard ML into the type theory developed by Lee et al. [2007], the "internal" language of the interpretation, equipped with a Plotkin-style structural operational semantics [Plotkin 1981]. A candidate Standard ML program is deemed well-formed exactly when it has a well-typed elaboration, and its execution behavior is defined to be that of its elaboration. Thus, the type safety of Standard ML is reduced to that of the internal language of the translation.

To facilitate mechanized metatheory the internal language is formulated with conventional binding and scope mechanisms that support capture-avoiding substitution by renaming bound variables. Its type system is inductively defined by inference rules that rely on implicit α-conversion to avoid identifier name conflicts. The type system accounts for the modularity mechanisms of Standard ML, including type abstraction and type sharing, and phase-respecting dependent types for modules, and includes recursive types, reference types, and dynamically allocated exceptions at the core language level. The internal language is equipped with a substitution-based structural operational semantics [Plotkin 1981] consisting of a state transition system defining the execution steps of a program and the propagation and handling of exceptions.

Type safety for the internal language is formulated as the conjunction of two properties, *preservation* and *progress* [Harper 2016; Wright and Felleisen 1991]. Preservation states that transitions from well-formed states are themselves well-formed; progress states that a well-formed state is either *completed*, or must either take a step or raise an exception. These properties ensure that *well-formed programs always go right*, a strengthening of Milner's formulation, *well-formed programs never go wrong*, that avoids the need for checks for ill-defined states in the operational semantics (which, after all, can never arise in the execution of a well-typed program).

Elaboration is defined by a relation between raw Standard ML program phrases and their internal language counterparts. The most important, and indeed indispensable, job of elaboration is *scope resolution*, which determines the binding and scopes of identifiers in a Standard ML program. Unlike the internal language, the scoping rules of Standard ML are surprisingly complicated, and can only be defined as part of its static semantics. For example, the identifiers introduced by an

open declaration can only be determined once the signature of the opened structure variable is determined.

Type inference is performed during elaboration by non-deterministically "guessing" omitted type information, much as it is in the Definition. Polymorphic generalization is represented using internal language analogoues of Standard ML functors that abstract over a structure of types. Equality polymorphism is handled by augmenting this structure with equality test functions on the equality type variables.[75] Datatypes and pattern matching are regarded as modes of use of modules. A datatype declaration is translated to the definition of a module defining an (opaque) abstract type whose operations correspond to the value constructors of the datatype and a case analysis function that is used in the elaboration of pattern matching. To account for the enrichment aspect of signature matching the elaboration generates a "forgetful functor" that coerces the candidate structure to the target signature.

The mechanization of type safety for Standard ML makes essential use of these distinctive capabilities of the Twelf proof asssistant [Pfenning and Schürmann 1999; Schürmann 2009]:

(1) It provides native support for *higher-order abstract syntax*, which codifies the uniformities of binding, scope, and substitution, eliminating the need to consider these explicitly in the metatheory.[76]
(2) It directly supports the *judgments as types* principle of the LF logical framework [Harper et al. 1993a]. The statics and dynamics of the internal language, and the elaboration relation, are codified in Twelf using this principle.
(3) It provides a *totality* checker that is used to verify the safety of the internal language, and to check that elaborated programs are well-typed.

The proof of safety is human-readable, written using pattern-matching in a similar way to which it is used in programming in Standard ML itself.[77]

The rôle of the Twelf totality checker in the verification of safety may not be immediately obvious. Briefly, Twelf permits the arguments of a relation (that is, the indices of a family of types) to be designated as "inputs" or "outputs." The totality checker ensures that the relation is total (every instance of the corresponding type family is inhabited). The relevance of totality checking to the verification of the safety of the language may be seen by examining its precise meaning:

(1) The preservation property may be viewed as stating that *for every derivation of a transition from one state to another* and *for every derivation of the well-formation of its starting state*, then *there is a derivation of the well-formation of its ending state.*
(2) The progress property may be viewed as stating that *for every derivation of the well-formation of a state*, then *there exists a derivation of that state being progressive*, in that it is either final, transitions, or raises an exception.

In each case the quantified assertion is a *judgment about the derivations of other judgments*, a characteristic feature of LF [Harper et al. 1993a]. Thus, preservation relates derivations of transition and of typing of the start state to a derivation of typing of the result state of the transition. Similarly, progress relates derivations of typing to either a derivations of transition, raising of an exception, or being a completed state. The quantifiers are expressed by mode declarations, and each case of the induction on inputs is populated by a constant. The correctness of each case is ensured by type checking, and the complete coverage of all cases is ensured by the totality checker. The same

[75] This account of equality polymorphism was later extended to Haskell-style type classes [Wadler and Blott 1989] by Dreyer et al. [2007].
[76] For a contrasting viewpoint, see Aydemir et al. [2008].
[77] See Crary and Harper [2009] for the complete mechanization in Twelf. The underlying theory on which it depends appears in the cited publications; the Twelf code is published only as the cited repository.

methods are used to prove that *for every derivation* that a Standard ML program elaborates *there exists a derivation* of well-typing of its elaboration, completing the safety proof.

7.3 Lessons of Mechanization

One purpose of the Definition was to support rigorous proof of the properties of the language, and in this regard it has been remarkably successful. Milner and Tofte's *Commentary on Standard ML* [1991] is an extensive development of a number of metatheoretic properties of the language, including the existence of *principal signatures* for program modules, a generalization of Milner's principal typing property for the functional core. Mechanical verification of such properties was anticipated, but at the time there were few provers that would be adequate for the task.

The first attempt to mechanize the safety of Standard ML was carried out by Elsa Gunter and Myra VanInwegen [VanInwegen 1996; VanInwegen and Gunter 1993] using the HOL prover [HOL 2019]. Although only partially successful in achieving its goals, this effort clearly demonstrated that a mechanized proof of safety was within reach, and called attention to the obstacles to a verified proof. The second attempt, described in the previous section, drew on this experience to achieve a mechanized proof of the safety of Standard ML using the Twelf prover.

Perhaps the biggest lesson of the Twelf mechanization is that it is essential to define a language hand-in-hand with the verification of its metatheory, much as in program verification it is essential to write programs with verification in mind. The separation of internal from external language separates scope resolution and type inference and management of derived forms from the formulation of its fundamental type structure and operational semantics. The safety of the internal language is stated crisply, without the need for instrumentation, and admits straightforward inductive proof, albeit with many cases. The Twelf prover makes it easy to verify that all cases are handled correctly (by type checking its formalization), and that all cases are covered correctly by a non-circular argument (by coverage and totality checking).

The critical step required to apply these methods to another language is to isolate a well-behaved internal language whose safety is easily stated and proved, and then to define the language in question by elaboration into the internal form. Absent such a formulation, it appears that the mechanical verification of the metatheory of a language remains a daunting task; we know of no other mechanization for a language of comparable scale to Standard ML.

8 THE SML BASIS LIBRARY

The *Standard ML Basis Library* specification, which was published in 2004 [Gansner and Reppy 2004], complements the *Definition (Revised)* by specifying a standard set of APIs that all SML implementations are expected to implement. This section details the history behind this part of the SML ecosystem, as well as describing some of the key aspects of the design.

8.1 A Brief History of the Basis Library

At the time of the publication of the Definition [Milner et al. 1990], there were five implementations of Standard ML at various stages of development (Edinburgh ML, MLWorks, Poly/ML, Poplog/ML, and SML/NJ). The existence of multiple implementations targeting the same language specification was a real success for the Definition, but it also exposed a major weakness, which was that the Definition did not specify a complete programming system. Specifically, it only defined a minimal *Basis* (described in Appendices C and D) of nine types and fewer than 50 operators and functions. There were glaring omissions, such as no `option` or `array` types and no `fold` functions over lists. Furthermore, the Basis did not take advantage of the module system; instead it consisted solely of core-language definitions. Thus, while these systems were implementing a common language standard, because of the minimal standard Basis, they differed significantly in the types and

operations that they provided. Differences in whether functions were curried or not and in the order of their arguments meant that even basic code written for one implementation was not portable to others.

In the Autumn of 1991, the SML/NJ implementors issued a proposal [Unknown 1991], titled "Toward a standard Standard ML," that began[78]

> We would like to begin seriously and systematically working on the problem of incompatible environments and libraries that is threatening to cause us all continuing grief and to retard the future development of Standard ML.

This document proposed defining a pervasive `Standard` structure, which would contain a set of bindings that all implementations would agree to provide. This first attempt to standardize a Basis for SML did not get much traction, in part because lumping everything into a single structure was unwieldy, but it did identify the problem and there continued to be discussions about it among the various implementors at conferences and workshops over the next few years. At the 1993 Functional Programming and Computer Architecture (FPCA) conference in Copenhagen, MacQueen proposed that the existing SML/NJ Basis be used as a starting point for designing a standard Basis Library specification that would replace Appendices C and D of the Definition. In response to this proposal, Dave Berry and Brian Monahan at Harlequin Ltd. wrote a detailed set of suggestions for revising the SML/NJ Basis toward adopting it as a common standard.[79]

The Harlequin proposal laid out guiding principles for what features should be covered by the Basis, as well as many specifics. The most important aspects included:

- A much richer set of primitive types and type constructors than previously defined. These included characters, multiple precisions of numbers, arrays, and vectors.[80]
- OS-independent operations on the filesystem.
- Support for more advanced I/O options, such as random access and both binary and text I/O.
- Support for networking based on the Berkeley Unix Sockets API.
- A design that used the SML module system to structure and organize the API.
- Support for internationalization, including support for differing character encodings and locales.

Over the course of the next six months, the Harlequin and SML/NJ teams collaborated on turning these proposals into a more complete and refined design. In March of 1994, a fairly complete draft of a Basis Library proposal was finished (titled *A New Initial Basis for Standard ML*) [Appel et al. 1994]. At this point, the scope of the Basis Library was beginning to come into focus. It included new types for characters and unsigned integers (called words); support for multiple precisions of numeric types; types for both polymorphic and monomorphic arrays and vectors; portable interfaces to system services; and richer support for I/O (but not yet networking). Furthermore, this document included several proposed language changes that were necessary to support the Basis. These included the addition of literal syntax for characters and words, overloading of numeric literals, and a syntax for vector literals in patterns and expressions.[81]

Over the next year or so, the Harlequin and SML/NJ teams collaborated on further refining the Basis Library design. This was a period of great activity that was further accelerated by Peter Sestoft's visit to Bell Laboratories in New Jersey from October of 1994 through June of 1995.[82] A

[78] The author of this document is uncertain, but it was probably David MacQueen.

[79] Unfortunately, this proposal was not dated, but based on file-system timestamps it was likely written in the Summer or Autumn of 1993.

[80] In SML terminology, arrays are mutable sequences, whereas vectors are immutable.

[81] The latter of these was implemented in SML/NJ, but was not included in the *Definition (Revised)*.

[82] Peter was one of the developers of Moscow ML, a version of SML that runs on top of the Caml-light bytecode interpreter.

number of key features were designed and added to the Basis Library during this period [Appel et al. 1995], including:

- Generic signatures (`MONO_ARRAY` and `MONO_VECTOR`) for monomorphic arrays and vectors.
- Organization of the Basis Library into modules so that every type and value had a home in some structure. Structures were defined for the builtin types (e.g., `Bool` and `List`).
- The addition of `substring` as a builtin abstract type. This idea was later generalized to all array and vector types as described in Section 8.7.
- A collection of conversion structures for converting between different precisions of numbers. This design was later abandoned as discussed in Section 8.4.2.
- A consistent design of iteration combinators for all aggregate types.
- A `Posix` structure that defined an SML interface to the IEEE Std 10031b-1993 standard [IEEE 1993]. This structure had substructures for each of the major components of the standard (e.g., `Posix.Process` for process management, etc.).

Up to early 1996, the work on the Basis Library was largely independent of the parallel effort to revise the Definition. This independence was because the Basis Library work was originally an effort to address limitations in the original Definition. With the *Definition (Revised)* nearing completion, however, there were several rounds of email between the Basis Library group and Mads Tofte, who was preparing the manuscript for the *Definition (Revised)*. While most issues were fairly easy to resolve, the question of support for multiple precisions of characters and strings required several iterations. The difficulty was largely because the Basis Library group had not figured out what the design for non-ASCII character sets was going to be (see Section 8.8 for more discussion).

While the *Definition (Revised)* was completed and published in 1997, it took another seven years to complete and publish the Basis Library specification. This lengthly gestation was in part because of the amount of material in the Basis Library. The final version had 67 distinct interfaces, many with multiple implementations and some with multiple sub-structures, and its published version [Gansner and Reppy 2004] has five times as many pages as the *Definition (Revised)*.

8.2 Design Philosophy

The scope of the Basis Library was an important question early in the design process. At the time, there were existing libraries that provided implementations of various abstractions, such as finite sets and maps, and common algorithms, such as sorting, but there was no standard interface for these. After discussion, it was decided to limit the scope of the Basis Library to features that are either broadly used (e.g., lists), or that require special compiler and/or runtime system support (this restriction was interpreted to include structures that provided operating-system portability, such as the `OS.Path` structure).

The most important design principle was that the module system would be used to organize the Basis Library (in contrast to the initial basis defined in the Definition, which defined everything at top level). This principle was first stated in the 1995 draft [Appel et al. 1995]:

> The initial basis is contained in a set of structures. Every type, exception constructor, and value belongs to some structure, although some are also bound in the initial top-level environment.

This requirement applied even to builtin types like `int` and `bool`. Using the module system to structure the Basis Library was an acknowledgment of the importance of modules in the design of the language and made it easy to extend the Basis with new types and operations without having to invent new globally-unique names. For example, the Definition defined the `length` function for computing the length of a list and the `size` function for getting the length of a string. The

module system allowed the addition of additional sequence types that could all use `length` (e.g., `Array.length` and `Vector.length`).

There was also an agreed upon set of orthographic conventions, such as using camel-case for functions and upper-case letters for data constructors (Chapter 1 of [Gansner and Reppy 2004]), although these were violated in a few cases where there was already significant prior use, such as `nil`, `true`, and `false`).

8.3 Interaction with the REPL

SML began life as an interactive language, and, thus, there was a strong tradition of working in a Read-Eval-Print-Loop (REPL) environment. Because of this tradition, there was a necessary design decision about what parts of the Basis Library would be available in the REPL without further user action.[83] For example, the Moscow ML implementation used a command-line switch to specify what modules would be available in initial REPL environment, and the Harlequin team argued that the choice of what was available in the REPL be left to implementations. The counter argument was that the purpose of having a Basis Library specification is consistency across implementations. This view eventually won out. Many SML implementations also support a "batch" compilation mode (in fact, some do not support a REPL). For these build systems, the user must specify inclusion of the Basis Library explicitly.

8.4 Numeric Types

Support for numeric types and operations in the Definition suffered from a number of problems. These included a paucity of numeric types, no specification of the semantics of numeric types, and design choices that made efficient implementation difficult. For example, the Definition defined a different exception for each arithmetic operator (i.e., `Quot`, `Prod`, `Sum`, and `Diff`). This design was overly complicated and precluded the use of hardware trapping mechanisms (because the hardware traps did not distinguish between different operations). In fact, some implementations, such as SML/NJ, defined these exceptions as aliases for a single exception. The Basis Library simplified this design by defining just two arithmetic exceptions: `Overflow`, for integer overflow, and `Div`, for integer division by zero.

8.4.1 Supporting Multiple Precisions. The Definition defined only two numeric types — `int` and `real` — of unspecified precision. There was clearly a need for multiple integer precisions (e.g., an implementation might want to support both small integers with fast arithmetic and arbitrary-precision integers, or both 32 and 64-bit floating-point numbers), but the design was complicated by the fact that existing implementations had already made different choices about representations. For example, SML/NJ and Moscow ML used a tagged 31-bit representation for the `int` type [Appel 1990], while Harlequin's MLWorks implementation used 30 bits.[84] The initial design [Appel et al. 1994] assumed that implementations would provide `SmallInt` (small, fixed-precision integers) and/or `LargeInt` (arbitrary-precision integers), with the `Integer` structure being an alias for one of these. In addition to integers, a `Word` structure was defined for fixed-precision unsigned integers. There was also a suggestion that an implementation might also provide other fixed-precision integer/word types in modules named `Int`*n* (resp. `Word`*n*) where *n* is the number of bits of precision. This initial proposal was fairly close to the final design, the main difference being a standardization on the `Int`*n*/`Word`*n* naming scheme with `IntInf` for arbitrary-precision integers. The same scheme was

[83] One must remember that computers were much slower and had significantly smaller memories at that time.
[84] The choice of 30 bits was motivated by the hardware support for tagged arithmetic on the SPARC processor.

used for real numbers (e.g., `Real64`).[85] Implementations were required to provide structure aliases for the default integer type (`Int`), default word type (`Word`), and the largest fixed-precision type (`FixedInt`).

This scheme was a compromise between the reality that existing implementations had made different choices about numeric representations and the need to support a way to write portable code. For most applications, one could program using the default types, but if the application required a specific precision, say 32-bit integers, then one could explicitly use the `Int32` module and be limited to those systems that provided it.

The choice to support both signed and unsigned integers at various precisions required some additional changes to the Definition. Word literals were prefixed with "`0w`" to distinguish them from integers and the *ad hoc* overloading of arithmetic operators was extended to numeric literals. It was also decided to specify default typings for operators and literals, so that definitions like

```
val n = 1;
fun inc x = x+1;
```

would only require explicit type annotations when a non-default numeric type was desired.

8.4.2 Conversions Between Types. Supporting multiple precisions of numeric types created a need for conversions between the different types. The early designs of the Basis Library defined a `Cvt` structure that contained five kinds of conversion substructures: i.e., conversions between different precisions of integer, word, and real types, plus conversions between integer and words, and integers and reals. A typical SML implementation might provide three precisions of integers and words (8, 31, and 32-bits) and a single real type, which required 16 different conversion substructures. It became clear that this approach was not going to scale well. In circa 1996, an alternative approach was adopted that takes advantage of the fact that one can promote numbers to the largest available representation without loss of information. Thus, a conversion from a 32-bit integer to an 8-bit integer could be expressed as the composition of two conversions.

```
val int32to8 = Int8.fromLarge o Int32.toLarge
```

With this design, the number of conversion functions defined by the Basis was linear in the number of numeric types, but there was a potential efficiency issue. To overcome this problem, Lal George and John Reppy developed a compilation technique that allowed for as efficient a solution as supporting direct conversions between every pair of numeric types. They defined a set of five primitive conversion operators that could express all of the conversions defined by the Basis:[86]

- TEST(n, m) — checked conversion from an n-bit two's-complement signed representation to m-bits where $n > m$.
- TESTU(n, m) — checked conversion from an n-bit unsigned representation to an m-bit two's-complement representation, where $n \geq m$.
- TRUNC(n, m) — truncate an n-bit value to an m-bit value, where $n \geq m$.
- EXTEND(n, m) — sign extend an n-bit value to a m-bit value, where $n \leq m$.
- COPY(n, m) — copy (i.e., zero-extend) an n-bit value to an m-bit value, where $n \leq m$.

This approach can handle arbitrary precision integers (i.e., the optional `IntInf.int` type) by allowing ∞ as a bit width.

In addition to these primitive conversion operations, George and Reppy defined an algebra of conversions that compilers could use to rewrite compositions of conversions into single operations.

[85] In retrospect, `float`, `Float64`, etc. would have been more accurate names, but `real` was backward compatible with the Definition.

[86] Note that this design deals only with conversions between integer and word types; conversions involving reals were less of an issue, since implementations were expected to have at most two different precisions of reals.

For example, the conversion from 8-bit words to 32-bit integers type could be defined by the following composition:

```
val word8toInt32 = Int32.fromLarge o Word8.toLargeInt
```

Assuming that the default integer size is not 32-bits and the large int type is `IntInf.int`, then this conversion would translate to

```
TEST(∞,32) o COPY(8,∞)
```

which can be rewritten to

```
COPY(8, 32)
```

In this way, a conversion between any two integer or word types can be simplified by the compiler to a direct conversion.

8.4.3 Real Numbers. By the mid 1990's, most computer hardware had adopted the IEEE 754 Standard for Floating-point arithmetic [IEEE 1985]. Therefore, it was decided that the Basis Library should specify IEEE Floating-point semantics for the `real` type. While this decision made sense from the point of view of wanting to support efficient floating-point computations, it raised a few interesting design problems.

The design philosophy of SML is biased toward safety over performance. If an arithmetic operation overflows its representation or is given invalid arguments, the usual response is raising of an exception. As processor clock speeds increased, this approach imposed significant performance penalties on floating-point computations. The IEEE Floating-point standard addressed this problem by introducing signed infinities and "Not-a-Number" (NaN) values into the floating-point representation.[87] While early implementations of SML used trapping floating-point arithmetic, it was clear that such semantics would prevent performant floating-point computation in SML. Therefore, it was decided to adopt the IEEE non-trapping semantics for the SML `real` type.

The decision to allow NaNs as real values in SML lead to the question of whether the `real` type should continue to support equality (as discussed in Section 4.5). The Harlequin team argued that `real` had always been an equality type in SML and that there was too much history to change it. Countering this argument was the fact that because the Basis Library specified non-trapping IEEE floating-point semantics, treating `real` as an equality type would break properties that users and implementors expected to hold. Specifically, the expression "`x = x`" evaluates to `false` when `x` is a NaN. This property also has implementation consequences. A common technique when implementing equality on data structures, such as lists, is to first check if the two objects are the same using pointer equality and only recursively compare them if the pointer-equality test returns false. This optimization is not valid if the lists might contain NaNs. Because of these technical problems, `real` was removed from the list of equality types and the `Real.==` function with IEEE 754 equality semantics was added as a replacement to the Basis. This change was also reflected in the *Definition (Revised)* by the removal of real literals from the syntax of patterns.[88]

8.5 Input/Output

Support for Input/Output (I/O) was a topic of discussion from the earliest design meetings in the 1980's. The first I/O design proposal was by Luca Cardelli, who proposed a single *bipolar* stream type [Cardelli 1983c]. At the June 1984 design meeting, it was recommended that the design for

[87] If an operation overflows the representation, the result is infinity, and if the result is not defined (e.g., `0.0/0.0`), the result is a NaN.

[88] Note that while pattern matching against a real literal does not have the problem of non-reflexive equality on NaNs, real literals are not always exactly representable as floating-point numbers, which could cause surprising behavior.

I/O follow Cardelli's proposal, but that unipolar streams should be used (i.e., separate input and output-stream types). This guidance led to a couple of competing proposals: one by Keven Mitchell and Robin Milner [Mitchell and Milner 1985], and another by Robert Harper [Harper 1986]. The latter proposed two I/O modules: `BasicIO`, which defined a minimal collection of I/O operations for text, supporting both terminals and text files, and `ExtendedIO`, which defined a few useful, but more complicated, operations, such as polling an input stream and running a Unix subprocess connected by I/O streams. The types and operations defined in Harper's `BasicIO` structure became the standard by being included in Version 1 of the Definition [Harper et al. 1987a].[89] His design was also supported by the early implementations: the Edinburgh SML implementation included the `BasicIO` structure and the earliest versions of SML/NJ provided an `IO` structure that merged Harper's two structures into one.

I/O support as provided by the Definition was quite minimal and only useful for simple text-based applications. It lacked support for random access in files, binary I/O, and unbuffered I/O. All of these were features that would be necessary to support richer applications.

In 1994, Andrew Appel and Reppy had a series of discussions about the design of a new I/O Library for SML/NJ. These discussions were focused on a layered design that was inspired by recent work on an I/O library for Modula 3 [Brown and Nelson 1989] and resulted in a series of eight iterations of the design, written by Appel, in response to feedback from Berry and others.[90] The final design involved three layers:

(1) The `PRIM_IO` interface defined reader and writer types that were an abstraction of unbuffered system I/O. Following the object-oriented design of the Modula 3 library, these types were defined as records of operations and included support for non-blocking I/O and random access.
(2) The `STREAM_IO` interface defined a I/O streams that provided buffered I/O. A key feature of the input streams was a functional semantics that allowed arbitrary look ahead.
(3) The `IO` interface (later renamed `IMPERATIVE_IO`) defined the traditional stream-I/O operations that had been in common usage.[91] Streams at this level were essentially mutable references to the streams from the `STREAM_IO` level, which allowed for redirection of I/O.

Each of these layers had two instantiations: one for binary I/O and one for text I/O. The `TextIO` stack included definitions for the standard input, output, and error streams.[92]

8.6 Sockets

It was clear that the Basis Library should include standard support for networking[93] and in late 1994, Berry submitted a proposal for adding sockets to the Basis.[94] Berry's proposal was based on the Berkeley UNIX and Microsoft's WinSock APIs, which were fairly standard at that point, and it was pretty much a direct mapping of the C-language API into SML; the two main differences were

[89] Since the Definition did not define any modules, these were included as top-level bindings.
[90] The first proposal was written in November of 1994 [Appel 1994] and the last in July of 1995 [Appel 1995]. These documents are available at http://sml-family.org/history.
[91] The names of the operations, however, were eventually changed to follow the camel-case conventions of the Basis Library.
[92] In retrospect, it was probably a mistake to include the standard I/O streams in the `TextIO` structure. An alternative approach would have been to put them in a `ConsoleIO` structure (and perhaps make them pervasive in the REPL environment). This would have made it easier to accommodate programs that are not connected to a TTY (e.g., GUI applications).
[93] The SML/NJ runtime included functions for creating client-side TCP/IP connections, which had been added to support the eXene X11 windowing library [Gansner and Reppy 1991, 1993], but there was no support for server-side or UDP/IP operations.
[94] This date is a bit of a guess; there is some evidence that Berry might have first proposed a sockets interface earlier, but there is no record of that proposal.

the use of abstract types to represent sockets, address families, etc., instead of the raw integers used in C, and the replication of the socket operations for each kind of socket. The latter difference ensured type security at the cost of a much larger number of functions in the API.

The major challenge in the design of the Sockets API for SML that Berry was addressing in his design was the problem of providing strong static types for socket operations. In the UNIX APIs, a socket is represented by an integer file descriptor, independent of its rôle or state. and dynamic checks are used to ensure that sockets are used correctly. For example, there is no static way to distinguish between a socket that is being used to accept stream connections and one being used to send datagrams.

Reppy responded to Berry's proposal with a design that improved the interface to network databases (for address lookup), added a `Poll` structure that generalized the `select` operation in Berry's design to also allow waiting for file I/O, and merged the distinct sets of socket operations into a single `Socket` structure. Berry objected to the single `Socket` structure because of the fact that having structures specialized to specific protocols and address families provided stronger static type guarantees (e.g., one could not mix up datagram and stream connections, because the socket types would be different).

The tension between the desire for a solution that provided fine-grain distinctions between different socket types and wanting to avoid a proliferation of different Socket modules lead to the discovery of a neat trick for encoding the distinctions using type constructors and distinguished abstract types as arguments to the constructors (a trick that was independently discovered by others and has come to be known as *phantom types* [Leijen and Meijer 1999]). This technique was used in Reppy's second design, where the `socket` type was replaced with the type constructor

```
type 'a sock
```

The type argument could be instantiated with an abstract type that represented the type and address family of the socket. This design allowed operations that did not depend on the type or address family to be polymorphic; e.g.,

```
val close : 'a sock -> unit      (* close a socket *)
```

while other operations were given monomorphic types

```
signature UDP_SOCK =
  sig
    type udp
    val socket : unit -> udp Socket.sock
    ...
  end
```

This design provided the type safety of Berry's proposal, but with a much smaller interface. Of course, it was possible to design an even better API by using two type variables: one for the address family and one for the "type" (datagram or stream). After three more iterations, the design was largely finalized with the following types:

```
type ('af, 'sock) sock
type 'af sock_addr
type dgram
type 'mode stream
type active
type passive
```

with phantom types (although that name had not yet been coined) being used to enforce type safety [Reppy 1996]. For example, the `accept` function, which waits for connection requests is given the type

```
val accept : ('af, passive stream) sock
      -> ('af, active stream) sock * 'af sock_addr
```

This type encodes the requirement that `accept` be called on an passive stream and the fact that it returns an active stream and the socket address of the connecting entity. Furthermore, the type requires that the address family of the argument socket be the same as the result socket and address.

8.7 Slices

The last major feature to be added to the Basis Library was the notion of a *slice* of an array or vector type, which was a generalization of the `substring` type.

The `substring` type and `Substring` structure were introduced to represent a subsequence of a string value.[95] The type and its operations were based on Wilfred Hansen's subsequence algebra [Hansen 1992]. The substring type encapsulates an underlying string value with a start index and length, which allows efficient *in situ* processing of strings. For example, tokenizing a string or removing leading and trailing whitespace could be done without copying the underlying string data.

Substrings proved to be a very useful mechanism, so in the Spring of 2000 MacQueen proposed generalizing them to all of the array and vector types in the Basis. For each array and vector module, a corresponding slice structure was added to the basis (e.g., `CharArraySlice`).

Slices provided an efficient way to do *in situ* processing of sequence data; for example, splitting a string into fields required copying the individual fields into new strings, whereas splitting a slice into fields just required defining a new slice for each field. Slices also provided a handy mechanism for iteration over sequences; both at the element level, where the slice represents the position of the next element, and as an argument to iteration combinators. Lastly, the addition of slices allowed a simplification of the array and vector signatures, since indexed versions of iteration combinators that specify the start and end of the iteration were removed in favor of equivalent operations on slices.

8.8 Internationalization and Unicode

Over the course of designing the Basis Library, there were a number of ideas that were explored in detail, but eventually removed from the specification. The most significant of these was support for internationalization and non-ASCII character sets. A follow-on to the 1993 proposal from Harlequin included a three-part discussion of how SML might support *locales* and non-ASCII character sets. This design proposal was fleshed out by Berry in May of 1995, with a design that added support for wide characters, and localization in the form of a `Locale` structure that was loosely based on the C-language localization mechanisms. These design ideas were incorporated in the 1995 drafts of the Basis Library specification.

Unlike most other parts of the Basis Library design, adding support for more character and string types required cooperation with the Definition effort, since the language syntax had to be extended to support non-ASCII characters.[96] As Tofte was wrapping up the *Definition (Revised)* during the spring and summer of 1996, there were several design iterations that attempted to address support for multiple character and string representations. In March 1996, Berry proposed a design with two classes of characters defined by top-level structures `Char8` and `Unicode`, with substructures for

[95] The `Substring` structure was added in the Spring of 1995.

[96] In fact, the `char` type and its literal syntax were ideas from the Basis effort that had already been added to the Definition.

strings and characters. The problem with this approach, however, was that the Definition was not structured to specify modules as part of its predefined basis; instead, it required all types to be defined at top-level. Therefore, Tofte proposed an alternative design that added the top-level types `unichar` and `unistring` to be the type of 16-bit Unicode characters and strings, with support for these types being required by all implementations.[97] The Basis design team pushed back against this proposal, because it did not allow for encoding schemes other than Unicode (e.g., Shift JIS) and because it required implementations to support two precisions of characters.

The resolution to this problem was to adopt the same approach to characters and strings as had been adopted for the numeric types and literals. The Definition specified two default, top-level, types (`char` and `string`) and implementations were allowed to support additional character types with the requirements that they provide at least 256 characters and agree with the ASCII character encoding on the first 128 values. As with numeric literals, character and string literals were overloaded when an implementation supported multiple types. A new escape sequence was added to specify characters outside the 8-bit range, since the syntax of SML programs was restricted to the ASCII character set. This design was included in the final draft of the Definition in late August 1996.

While the question of how to handle characters and strings had been resolved, there were various related issues, such as how to specify conversions between character encodings, how to support variable-length encodings (e.g., UTF-8), and what kind of locale-specific support should be provided. In the end, the design committee was uncertain about the best way to handle internationalization and so took a conservative approach with the expectation that a future extension of the Basis Library could handle the problem. As a result, the `Locale` structure was removed from the library and the only provision for non-ASCII characters was the definition of a `TEXT` signature and an optional `WideText` structure that could be a home to a multibyte representation of characters and strings.

8.9 Iterators

Another example of a major feature that did not make the final cut was a general framework for iteration. At one point, there was a design proposal for an iteration abstraction (essentially a stream type) and that there would be generators for every sequence type. This design then allowed a single set of iteration combinators (e.g., `map`, `fold`, etc.) to be defined. While this design was elegant and would have improved modularity and code reuse, there was concern that the implementation overhead of the iteration abstraction would be too great, so it was rejected.

8.10 Publication and Future Evolution

The Basis Library specification was finalized in 2003 and published as a book by Cambridge University Press in 2004 [Gansner and Reppy 2004]. As part of the publication agreement, the "manual pages" part of the specification[98] was posted on the web in HTML and permission to redistribute these pages was provided on request. The decision to publish the Basis Library as a book, rather than just posting it on the web was partially driven by a belief by some stakeholders that the existence of a book would give SML extra credibility. The decision was also a function of the times; web distribution of documentation was not very widespread in the mid 1990s.

Unlike the case of the published Definition of Standard ML, the expectation was that the Basis Library would evolve to reflect changes in systems and practice. For a number of years, however, the Basis did not change, since implementors were more focused on compilers, higher-level libraries,

[97] This design was contained in a June 1996 draft of the Definition.
[98] The published book also included seven supplementary chapters of tutorial material.

and applications. In the last few years, however, a number of changes to the Basis Library have been proposed and integrated into some implementations.[99]

8.11 Retrospective

It most respects, the Basis Library standardization achieved its desired goals; it provided a foundation for applications and higher-level libraries to be written and easily ported across multiple SML implementations. There are a few areas, however, where the results might have been better:

- The handling of different integer and word sizes, while necessary to bridge differences in existing implementations, is inelegant. In retrospect, it probably would have been better to specify a small collection of standard sizes that would be present on all implementations.
- Another mistake in the design of numeric types is the fact that the default types are aliases of implementation-dependent types instead of being abstract. For example, `int` is an alias for `Int31.int` in some implementations, while it is `Int32.int` in others. Code that inadvertently relies on such aliasing will be non-portable. If the default types were abstract, then such portability bugs would be caught by the type checker.
- Neither the Definition nor the Basis Library addressed the question of interoperability with foreign code. As a result, libraries and applications that build on foreign code are non portable. Differences in the compilation models make specifying a standard foreign-function interface mechanism difficult. For example, batch-compiler implementations like the MLKit compiler or MLton can statically link to foreign functions, whereas interpreters like MoscowML or interactive compilers like SML/NJ must use dynamic loading and linking. Matthias Blume's NLFFI design [Blume 2001] provides some portability, since it is supported on multiple implementations, but there are still differences in linking.
- While the Basis Library provided a standard API for writing SML code, it did not specify any requirements for how to specify the building and linking of libraries and applications. As discussed in Section 5.7, the Definition did not address mechanisms for separate compilation, which are a necessary prerequisite for a scalable build system. Scaling to large systems requires a graph structure and visibility control mechanisms at the library level. Unfortunately, there are several different, and incompatible, solutions to specifying the composition of compilation units [Blume and Appel 1999; Cejtin et al. 2004; Harper et al. 1994; Swasey et al. 2006a], which means that to write a portable SML library or application requires writing multiple build specifications.

8.12 Participants

Many people contributed to design and specification of the Basis Library specification. The main design work in the period between FPCA '93 and the completion of the *Definition (Revised)* was carried out by the implementors of Harlequin's MLWorks system, SML/NJ, and Moscow ML: Andrew Appel, Matthew Arcus, Nick Barnes, Dave Berry, Richard Brooksby, Emden Gansner, Lal George, Lorenz Huelsbergen, David MacQueen, Brian Monahan, John Reppy, Jon Thackray, and Peter Sestoft.

The final document was edited by Emden Gansner and John Reppy, who were both involved in the design process from the start. The other principal contributors were Andrew Appel, Dave Berry, and Peter Sestoft, with additional design and writing contributions from Nick Barnes, Lal George, Lorenz Huelsbergen, David MacQueen, Dave Matthews, Carsten Müller, Larry Paulson, Riccardo Pucella, Jon Thackray, and Stephen Weeks. Furthermore, countless members of the SML community provided feedback on various iterations of the design over the years.

[99] These proposals have been gathered at https://github.com/SMLFamily/BasisLibrary.

9 CONCLUSION

To wrap up the history of Standard ML, we review mistakes that were made in its design, its place in the larger world of programming languages, and its post-definition history.

9.1 Mistakes in the Design of Standard ML

Following the publication of the Definition of Standard ML in 1990, there were a number of critiques written about the language. As part of developing the definition of EML (see below), Stefan Kahrs identified a number of errors in the Definition in his detailed (and occasionally pedantic) examination of the Definition [1993]. Andrew Appel [1993] and David MacQueen [1994] wrote more reflective critiques that covered both the strengths and weaknesses of the language. The issues raised by these critiques and other discussions were part of the motivation for the ML2000 effort described in Section 3.3.1 and for the *Definition (Revised)*.

While the *Definition (Revised)* fixed many of the detailed issues raised by Kahrs as well as some of the larger issues, such as imperative type variables, the final design was not perfect. Andreas Rossberg compiled a catalogue of defects in the *Definition (Revised)* [Rossberg 2013a]. While many of these were minor issues in the writing of the definition, Rossberg observed that the Definition was ambiguous in several important places. For example, the specification of overload resolution is not precisely defined, which leads to incompatibilities between implementations.[100]

In addition to the technical defects identified by Rossberg and others, there were a number of significant omissions from the Definition. Surprisingly, the Definition's specification of the language's syntax is lacking in many ways. There is no specification of the full syntax (many syntactic features are defined as derived forms) and the grammar is ambiguous in a number of places. The lack of a standard code-assembly mechanism and the lack of a standard foreign-function mechanism are two other serious omissions that were already discussed in Section 8.11.

9.2 The Impact of Standard ML

Standard ML was the first statically-typed functional language to gain broad use ranging from the class room to research projects to commercial applications. It provided a vehicle for research in programming-language design, semantics, and implementation that resulted in many research papers and dissertations. There are over a dozen textbooks based on SML, including introductory programming texts [Felleisen and Friedman 1998; Paulson 1996; Ullman 1998], concurrent programming [Reppy 1999], compilers [Appel 1998], and research monographs [Appel 1992; Nielson 1997]. There are also a number of substantial on-line texts about SML programming [Gilmore 1997; Harper 2011; Pucella 2001; Shipman 2002].

9.2.1 Applications of SML. The original application of ML was the Edinburgh LCF system [Gordon et al. 1979], which had a strong influence on the original design of ML [Gordon et al. 1978a]. In particular, the LCF/ML's abstract type mechanism was designed to allow only provable theorems (which were values of an abstract type) to be constructed. Likewise, the need to implement proof tactics motivated the inclusion of higher-order functions and failure trapping. The subsequent design evolution from LCF/ML to Standard ML was not driven by applications (other than its use in proof systems), but once there were a number of robust implementations of Standard ML available, the language became an attractive tool for implementing other systems. While these applications were largely developed after the definition was finalized, and thus did not influence the design, they did have some influence on the design of the SML Basis Library and on implementations of SML.

[100] Kahrs identified problems with overloading in the Definition [1993], but these were not fixed in the *Definition (Revised)*.

Because of its roots in proof systems, SML has proven to be particularly well suited to applications that manipulate syntax trees of one form or another. These, of course, include a number of proof assistants; the most notable being HOL [HOL 2019], Isabelle [Isabelle 2019], LEGO [Luo and Pollack 1992], and Twelf [Pfenning and Schürmann 1999; Schürmann 2009]. Some implementations (e.g., Moscow ML and SML/NJ) have included features, such as quote/antiquote mechanisms and customizable pretty printers, to better support proof systems. Related to theorem provers is *The Concurrency Workbench* [Cleaveland et al. 1993], for reasoning about concurrent systems specified using CCS.[101]

Compiler implementations share many characteristics with proof assistants, so it is not surprising that the most popular use of SML has been for writing compilers.[102] We discuss the many SML compiler projects below in Section 9.2.3 and the Appendix. Other examples include Moby [Fisher and Reppy 1999, 2002], which explored the combination of class-based object-oriented programming with the ML module system; Nova [George and Blume 2003], which was a domain-specific language for programming network processors, PolyMLR [Blume et al. 2006, 2008], which explored a generalization of Atsushi Ohori's polymorphic record calculus; and Diderot [Chiw et al. 2012], which is a parallel domain-specific language for scientific image analysis and visualization.

Another common area of application is in various kinds of program analysis tools: Aiken et al. implemented the *Berkeley Analysis Engine* for constructing constraint-based program analyses using SML [Aiken et al. 1998], which was used to build a system for detecting Y2K bugs in C code [Elsman et al. 1999]. *AnnoDomini* was a commercial source-to-source conversion tool for eliminating Y2K bugs from COBOL programs [Eidorff et al. 1999].

Database software is another, perhaps unexpected, area where SML has found application. Timothy Griffin and Howard Trickey at AT&T Bell Laboratories developed a tool called *pdiff* for writing *safe* transactions[103] on the embedded relational database in AT&T's 5ESS telecommunications switch [1994]. Other examples from the world of databases includes the Kleisli Query System [Wong 2000] and Chlipala's Ur/Web system for dynamic web applications with SQL backends [2015].

While most of the above examples are applications that have term manipulation as a core component, SML has also been used for a number of more traditional systems projects and applications. Gansner and Reppy developed the eXene GUI library [1991; 1993] using the Concurrent ML extensions in SML/NJ [Reppy 1991, 1999]. This library had a complete client-side implementation of the X11 network protocol and was used in a number of other projects. The work on eXene was a major motivator for the sockets APIs in the Basis Library (Section 8.6). The FoxNet project at CMU pushed the use of SML as a general systems programming language [Harper and Lee 1994]. A particular focus of the project was the use of the SML module system to build protocol stacks from reusable microprotocol implementations [Biagioni et al. 1994]. The use of SML for performance-critical code, such as protocol stacks, was a major motivation for the subsequent work on the TIL and TILT compilers, which are described below. In addition to the Ur/Web system, there are a couple of other examples of SML being used to implement web servers. The Swerve web server was developed by Anthony L. Shipman using Concurrent ML [Shipman 2002]; this system was later ported to MLton. And Elsman implemented an Apache Web server plugin using the ML Kit compiler that allows dynamic web applications to be implemented in SML [Elsman and Hallenberg 2003]

9.2.2 Language Design. Standard ML popularized a number of features that functional-language programers take for granted today. The most important of these being datatypes with pattern matching, Hindley-Milner type inference, and parametric polymorphic (or "generic") types. In

[101] This tool lives on as the *Edinburgh Concurrency Workbench* (http://homepages.inf.ed.ac.uk/perdita/cwb).
[102] In fact, MacQueen has described SML as "a domain-specific language for compilers."
[103] Safe transactions are ones that maintain the integrity constraints of the database.

the latter case, Standard ML has arguably played a key rôle in popularizing the notion of type inference and of parametric polymorphic (or "generic") types in modern programming languages. For example, Andrew Kennedy's experience with implementing SML on the JVM was directly transferred to Microsoft's .NET environment [Kennedy and Syme 2001].

While the definition of SML has been static since 1997, the language has been used as the host for many experiments in language design. These include extensions to pattern matching [Aitken and Reppy 1992], non-local control flow [Benton and Kennedy 2001; Harper et al. 1993b], extensions to the type system, such as adding dimension types [Kennedy 1996] and record polymorphism [Ohori et al. 2014], and extending the language to support staged meta programming [Taha and Sheard 2000].

SML's module system is one of it most distinguishing features and directly influenced the design of modules in Caml Special Light [Leroy 1994]. The design and semantics of modules has also inspired many research efforts. These include generalizing modules to support higher-order functors [Dreyer et al. 2003; Harper and Lillibridge 1994; Leroy 1995; MacQueen and Tofte 1994; Russo 1998], first-class (or dynamic) modules [Rossberg 2015; Rossberg et al. 2015; Russo 1998], recursive modules [Crary et al. 1999; Dreyer 2007; Dreyer et al. 2001; Russo 2001], mixin modules [Dreyer and Rossberg 2008; Duggan and Sourelis 1996; Hirschowitz and Leroy 2005], and modular type classes [Dreyer et al. 2007]. There has also been a recent line of research to unify the notion of structures and signatures [Rossberg and Dreyer 2013], and to unify the core and module languages [Rossberg 2018].

Extended ML (EML) was a language that combined SML's structures and signatures with algebraic specifications [Kahrs and Sannella 1998; Sannella and Tarlecki 1985]. The goal of EML was to allow a program to be developed in steps from an unimplemented algebraic specification to a complete program. EML extended the SML module language by allowing axioms to be included in signatures, where they specify the behavior of operations, and in structures, where they serve as placeholders for unimplemented operations. Likewise, the formal definition of EML was an extension of the Definition of Standard ML [Kahrs et al. 1994, 1997] and its development resulted in the identification of many issues in the Definition [Kahrs 1993]. EML was designed to be a front end that would produce proof obligations to be solved by a supporting theorem prover. Front ends were built based on the ML Kit compiler and, later, on Moscow ML, but there was never a complete system.

9.2.3 Language Implementation. One of the most significant impacts of SML has been the exploration of a wide and diverse range of implementation techniques in SML compilers.

The Standard ML of New Jersey (SML/NJ) implementation built on previous work in the Scheme community to push a continuation-passing-style IR to its limit [Appel 1992; Appel and Jim 1989]. This IR was pushed further with the development of a technique to model callee-save registers in the IR [Appel and Shao 1992] and a "safe-for-space" closure conversionalgorithm [Shao and Appel 2000]. The SML/NJ system provided a testbed for many other lines of research in language implementation research. These include Andrew Tolmach's work on replay debugging [1992] (which exploited SML/NJ's heap-allocated continuations to achieve space and time efficient checkpointing) and the MLRISC code generation framework [George et al. 1994], which was originally developed to provide a portable backend for SML/NJ, but became a platform for research into register allocation and other code generation techniques [Appel and George 2001; George and Appel 1996; Leung and George 1999]. Nick Benton, Andrew Kennedy, and others developed the MLj compiler that compiled SML to the Java Virtual Machine [Benton and Kennedy 1999; Benton et al. 1998]. They followed this project with an SML compiler for Microsoft's .NET platform [Benton et al. 2004]. The SML.NET compiler used an imperative implementation of its IR to implement linear-time

shrinking [Benton et al. 2005] and used a limited form of continuation in its IR to model loops and other control flow [Kennedy 2007].

Several SML compilers have explored whole-program compilation techniques. One interesting line of research has been the use of monomorphization,[104] where a polymorphic program is converted to a first-order monomorphic one by instantiation and defunctionalization [Reynolds 1972]. The combination of these techniques allows the compiler to get precise information about control flow and runtime representations. The RML compiler used this approach to compile SML to Ada [Tolmach and Oliva 1998], while the MLton compiler exploits these techniques to produce highly-optimized code [Cejtin et al. 2000].

The use of advanced type systems for compiler intermediate representations was pioneered in Standard ML compilers; particularly the *Typed Intermediate Language* (TIL) work at CMU [Morrisett 1995; Petersen 2005; Tarditi 1996; Tarditi et al. 1996] and the FLINT IR that Zhong Shao and his students developed for the SML/NJ compiler [Saha and Shao 1998; Shao 1997; Shao and Appel 1995; Shao et al. 1998]. The TIL compiler emphasized the use of intensional type analysis and run-time dispatch on types to compile polymorphism; the TILT compiler [Petersen et al. 2000; Petersen 2005] extended these methods to the full language, including modules and separate compilation [Swasey et al. 2006a]. The development of these compilers inspired, and was inspired by, the reformulation of the Standard ML definition described in Section 7 and led to other research on the use of types in intermediate languages [Morrisett et al. 1998].

The Church project was a compiler for SML that used a typed intermediate language, CIL, which allowed polyvariant flow properties to be expressed using intersection types [Wells et al. 2002]. This information could be used to support optimizations, such as specializing the representation of function values [Dimock et al. 2001].

Runtime systems and, especially, garbage collection (GC) have also been an active area of research in the SML community [Appel 1987, 1989b, 1990; Appel and Concalves 1993; Reppy 1993]. A particularly interesting line of research explored using static type information to eliminate runtime tags [Appel 1989a; Goldberg 1991; Goldberg and Gloger 1992; Tolmach 1994]. This work was based on the observation that it was possible to reconstruct the types of values as the garbage collector traced the live data by using the static typing context for the GC point. Tolmach also used these type reconstruction techniques in his replay debugger [1992].

One of the most influential ideas to come out of the SML implementation community is the use of region-type inference to manage memory without garbage collection [Tofte and Birkedal 1998; Tofte et al. 2004; Tofte and Talpin 1994, 1997]. This technique uses an effects type system to associate a *region* with every allocation. The static region information is used to guide an arena-allocation strategy [Hanson 1990], where all of the memory in a region is reclaimed in a constant-time operation when the data in the region is guaranteed to be unreachable. Supporting region types, which were internal to the compiler, with separately compiled modules required new techniques for propagating information across compilation boundaries and for inter-module optimization [Elsman 1999a]. These techniques were implemented in the ML Kit compiler [MLKit 2019].

9.2.4 Concurrency and Parallelism. SML has provided the base for significant research in concurrent and parallel language design and implementation going back to its earliest days. For example, Kevin Hammond's Ph.D. research explored the compilation of SML Version 1 [Harper et al. 1987a] to a parallel graph-reduction machine [Hammond 1988].

The earliest support for concurrency in SML was the `Process` module provided by the Poly/ML implementation, which supported synchronous message passing over typed channels. This design

[104] Monomorphization is possible in SML because the language does not support first-class polymorphism or polymorphic recursion.

was influenced by Holmström's PFL [1983] and Milner's Calculus of Communicating Systems [1980]. It included a non-deterministic choice operator that forked two subprocesses, but would only allow one of them to communicate. In the general case, this mechanism was an *unguarded choice* that was challenging to implement. Independently, Reppy developed a message-passing framework for the Pegasus ML language (an ML dialect) [1988] that provided an abstraction mechanism for synchronous communication. The SML/NJ implementation added support for first-class continuations around 1989, which made experimenting with user-space concurrency mechanisms quite easy. At Princeton, Norman Ramsey developed a message-passing library using SML/NJ's continuations [1990] and, at Cornell, Reppy reimplemented the mechanisms from Pegasus ML in a language that he called *Concurrent ML* (CML) [Reppy 1989, 1991, 1999]. Cliff Krumvieda developed distributed programming constructs based on CML and the Isis atomic multicast model [Cooper and Krumvieda 1992; Krumvieda 1993].

There were also more traditional shared-memory concurrency libraries developed for SML. Morrisett and Tolmach developed a multiprocessor extension of SML/NJ [Morrisett and Tolmach 1993] and eventually Poly/ML replaced its `Process` module with a `Threads` module that implements a PThreads-style programming model.

More recently, there have been several projects on building scalable parallel programming languages using SML as sequential core. The Manticore project developed a parallel language, called Parallel ML (PML), starting with a reference-free subset of SML, to which lightweight parallel constructs were added (e.g., parallel tuples, comprehensions, and bindings) [Fluet et al. 2010]. PML also provided explicit concurrency based on a parallel implementation of the CML primitives [Reppy et al. 2009]. The MLton SML compiler has been used as the base for a couple of projects exploring parallel programming [Jagannathan et al. 2010; Westrick et al. 2019]. Various researchers have also explored parallel garbage-collection techniques in SML implementations [Auhagen et al. 2011; Cheng and Blelloch 2001; Sivaramakrishnan et al. 2014].

9.3 The Non-evolution of SML

The fact that the design of Standard ML has been frozen in time probably contributed to a decline in its popularity in recent years. The reason for this stasis can be traced back to a debate that occurred in 2001 — after the *Definition (Revised)* had been published, but while there was much activity on SML implementations and on the Basis. At that time, there was discussion among many of the SML implementors about ways to fix mistakes in the *Definition (Revised)* and to standardize useful extensions to the language. This discussion prompted a message on September 25, 2001 from Robin Milner and Mads Tofte to the `sml-implementors` list (hosted at sourceforge.net) that stated

> . . . we will not accept further revision of Standard ML. It is natural that people discuss changes and new features, and thus that successors of the language will arise. We expect, however, the designers of any successor language to make it clear that their language is not Standard ML.

This message led to a vigorous debate about the tension between preserving the stability of the language and the need to iteratively improve it.[105] While there are many (perhaps a majority) who argued for continuing evolution of the language design, Milner and Tofte remained unconvinced.

Fundamentally, the problem was that the only way that the language could evolve was by changing the Definition, but the Definition was a physical book that required agreement by its authors and major effort to update (e.g., the *Definition (Revised)*). There was no institution, such as a standards committee, that could maintain and update the Definition. Because Milner held the

[105] It should be noted that the authors were all in favor of continued evolution of the language.

copyright on the Definition and because the TEX sources for the Definition were not available to the community, there was to be no further evolution of the language.

As an alternative, an effort was started to design *Successor ML* as a collection of patches to the revised Definition.[106] These patches ranged from minor syntactic generalizations, to core-language extensions (e.g., pattern guards and syntax for functional update of records), to module-language extensions. Rossberg eventually created a version of his HaMLeT implementation of SML that supported the Successor ML patches [2008]. More recently, the MLton and SML/NJ implementations have started to support some of the core-language extensions of Successor ML.

9.4 Summary

While the Standard ML language has not changed over the past 20+ years, there continue to be at least five actively supported implementations with an active user community. There are a number of widely used systems, such as the Isabelle/HOL theorem prover, that are implemented in SML and there are many research systems that use SML.

Standard ML occupies a "sweet spot" in the design space; it is flexible and expressive while not being overly complicated — in contrast with many functional languages (e.g., Haskell, OCaml, and Scala) that have very rich, but complicated, type systems, and so many features that most programmers use only a subset of the language. SML's elegance and simplicity is especially a virtue in education, providing an ideal vehicle for teaching fundamental concepts. It has also proven to be an effective tool for research, particularly in compilation, verification, and theorem proving, to which its expressive power is ideally suited. In both teaching and research, its stability allows its users to focus attention on their own work and avoid the distractions that arise when a language is defined by an ever-changing implementation.

ACKNOWLEDGMENTS

The authors would like to acknowledge the invaluable help of Mike Gordon and Peter Sewell, who scanned Robin Milner's files on Standard ML. Andreas Rossberg provided a very helpful literature survey of the more recent theoretical work on module systems. Stephanie Balzer, Karl Crary, Martin Elsman, Matthew Fluet, Tim Griffin, Karen MacQueen, Claudio Russo, Don Sannella, Peter Sestoft, and Kartik Singhal proofread drafts of this paper and provided useful feedback. Lastly, we thank the HOPL IV referees for their many detailed comments and suggestions and Kim Bruce, in particular, for his help as the shepherd for the paper.

A STANDARD ML IMPLEMENTATIONS

The existence of a formal definition of Standard ML has encouraged many implementations of the language. These projects have had differing goals and have made various different implementation tradeoffs, but they all essentially implement the same language. In this appendix, we provide a brief survey of these efforts, listed in alphabetical order. We restrict this list to implementations of the Standard ML and do not include related systems such as Manticore, Multi-MLton, or SML#.

A.1 Alice

Alice is a functional language developed at Saarland University as a sequel to the Oz language [Alice 2014; Rossberg et al. 2006]. The design of Alice is based on Standard ML, but adds extensions to support concurrent, distributed, and constraint programming. Alice runs on a virtual machine that

[106] This effort was organized as a (now defunct) wiki; an archive of this wiki can be found at http://sml-family.org/successor-ml/OldSuccessorMLWiki/Successor_ML.html or at https://web.archive.org/web/20190819224053/http://sml-family.org/successor-ml/OldSuccessorMLWiki/Successor_ML.html.

supports a completely machine independent execution model, which allows code to be migrated at run time between machines with different architectures.

A.2 Edinburgh ML

Edinburgh ML was the first implementation of the language. During the SML design period it was gradually modified to serve as an approximate prototype implementation of Standard ML and it served as the initial development environment for Standard ML of New Jersey. It was a reimplementation of Cardelli ML written in itself (with the runtime system written originally in VAX assembler and later rewritten in C) [Mitchell 1983; Mitchell and Mycroft 1985], and was developed by Alan Mycroft and Kevin Mitchell in the LFCS at Edinburgh, with later contributions by John Scott.

A.3 HaMLeT

HaMLeT is a reference implementation of Standard ML that was developed by Andreas Rossberg [2013b]. The implementation attempts to follow the structure of the Definition as closely as possible. It consists of a front-end that implements the statics and an interpreter that implements the dynamics. It also includes a simple JavaScript code generator. One of the motivations for HaMLeT is to support experiments in language design; to this end, it was used to implement the proposed Successor ML features [2008].

A.4 The ML Kit

An effort begun at Edinburgh by Tofte and Turner and others to build an implementation that was clearly derived from the Definition as a reference compiler [Birkedal et al. 1993; MLKit 2019]. The ML Kit compiler, which is actively maintained by Martin Elsman, uses a memory manager based on region inference combined with a tracing garbage collector [Hallenberg et al. 2002]. It is used as the front end for the SMLserver [Elsman and Hallenberg 2003] and to implement SMLtoJs (pronounced "SML toys"), which is an SML compiler that runs in a web browser [Elsman 2011].

A.5 MLton

The MLton implementation of Standard ML started as a source-to-source defunctorizer for SML/NJ written by Stephen Weeks [MLton-history 2014] in April 1997. The performance benefits for functorized code motivated the start of a project to write a new SML compiler that used advanced static analysis and aggressive whole-program optimization techniques. The resulting compiler provides perhaps the best performance of any functional language compiler across a wide range of benchmarks. The MLton implementation is still actively used and is maintained by Matthew Fluet [MLton 2020].

A.6 MLWorks

Harlequin's MLWorks was a commercial integrated development environment for SML that provided a rich set of graphical development tools [Haines 1991]. It is no longer supported, but its source code is available on GitHub (https://github.com/Ravenbrook/mlworks).

A.7 Moscow ML

Moscow ML is a light-weight implementation of SML that uses the Caml Light bytecode interpreter and runtime system. Version 1.00 was created in 1994 by Sergei Romanenko at the Russian Academy of Sciences and Peter Sestoft in Copenhagen. Later contributors include Ken Friis Larsen and Niels Kokholm. Claudio Russo added the full Standard ML module system (with some extensions) in 2000, based on the theory developed in his PhD dissertation [1998]. Originally intended mostly to support

a good teaching experience, Moscow ML has subsequently been used to build the HOL4 proof assistant system, comprising several hundred thousand lines of code, and many other research projects. Moscow ML is still maintained and is available from https://mosml.org.

A.8 Poly ML

Poly/ML was developed by David Matthews at Cambridge University; it gets its name from the fact that the original implementation was written in the experimental language Poly [Matthews 1988], although more recent versions are written in SML [Matthews 1989]. For a while, Poly/ML was licensed to Abstract Hardware Limited who used it to build hardware verification tools, but Poly/ML has been open source since 1999 and is available from https://www.polyml.org.

A.9 Poplog/SML

The Poplog system developed at Sussex University in the early 1980s was an implementation of the Pop-11 language (a descendant of Pop-2 [Burstall and Popplestone 1968]) that included support for Prolog (hence the name). Subsequently, support for Common Lisp and later Standard ML was added to the implementation; the SML implementation was owed to Robert Duncan and Simon Nichols and corresponded roughly to the 1990 version of the language. While SML support still appears to be part of the Poplog system, its support for the language was never updated to the 1997 version [POPLOG-SML 2019].

A.10 RML

The RML compiler, developed by Tolmach and Oliva [1998], was a compiler that translated SML to Ada. Similar to the MLton compiler, which was being developed around the same time, the RML compiler used whole-program monomorphization and defunctionalization, although its motivation was to support interoperability with Ada (whereas the MLton compiler used these techniques to enable aggressive optimization).

A.11 MLj and SML.NET

The MLj compiler was developed by Benton, Kennedy, Russell and at Persimmon IT's Cambridge research group in the late 1990's and compiled Standard ML to Java Virtual Machine (JVM) byte-code [Benton et al. 1998]. The MLj compiler is no longer supported, but its source code is available from http://www.dcs.ed.ac.uk/home/mlj.

Subsequently, Benton and Kennedy moved to Microsoft Research in Cambridge England, where they ported the MLj compiler to target Microsoft's .NET platform [Benton et al. 2004]. This compiler, called SML.NET, could be integrated into the Visual Studio IDE. The SML.NET compiler is also no longer supported, but its source code is available at https://www.cl.cam.ac.uk/research/tsg/SMLNET.

A.12 Standard ML of New Jersey

Standard ML of New Jersey was originally developed by MacQueen and Appel at Bell Labs, Murray Hill and Princeton University (hence the New Jersey) starting in the spring of 1986. The project was an entirely fresh implementation whose aim was to produce high quality native code for efficient execution. The compiler was written in Standard ML (after an initial, pre-bootstrap phase where the implementation language was Edinburgh ML enriched with SML features). It initially produced native machine code for VAX and Motorola 68000 family processors, but was later ported to the SPARC, IBM RS6000 (Power), HPPA, Intel x86, and DEC Alpha, running under various flavors of Unix, Microsoft Windows, and macOS. It is still actively supported by MacQueen and Reppy and is available from https://smlnj.org.

A.13 SML#

SML# is a dialect of Standard ML that generalizes the labeled record types of SML to support a form of record polymorphism [Ohori 1992, 1995]. The language was developed by Atsushi Ohori and his students. The implementation is actively maintained and is available from https://www.pllab.riec.tohoku.ac.jp/smlsharp.

A.14 TIL and TILT

The TIL SML compiler was developed at Carnegie Mellon University by Robert Harper, Peter Lee and their students [Morrisett 1995; Tarditi 1996; Tarditi et al. 1996]. This compiler was novel in that it used a strongly typed intermediate language to improve the efficiency of generated code for polymorphic languages. The TILT (TIL Two) compiler [Petersen et al. 2000] extended the methods of TIL to the full SML language and added the feature of generating typed assembly code [Morrisett et al. 1998]. While TIL and TILT are no longer supported, the TILT source code can be found at https://github.com/RobertHarper/TILT-Compiler.

REFERENCES

Alexander Aiken, Manuel Fähndrich, Jeffrey S. Foster, and Zhendong Su. 1998. A toolkit for constructing type- and constraint-based program analyses. In *Proceedings of the 2nd International Workshop on Types in Compilation (TIC '98)* (Kyoto, Japan) (*Lecture Notes in Computer Science*), Vol. 1473. Springer-Verlag, New York, NY, USA (March), 78–96. https://doi.org/10.1007/BFb0055513

William E. Aitken and John H. Reppy. 1992. *Abstract Value Constructors: Symbolic Constants for Standard ML.* Technical Report TR 92-1290. Department of Computer Science, Cornell University (June). https://people.cs.uchicago.edu/~jhr/papers/1992/tr-sml-const.pdf Archived at Internet Archive: https://web.archive.org/web/20170715041149/https://people.cs.uchicago.edu/~jhr/papers/1992/tr-sml-const.pdf (15 July 2017 04:11:49) A shorter version appears in the proceedings of the "ACM SIGPLAN Workshop on ML and its Applications," 1992.

Universität des Saarlandes 2014. *Alice.* Universität des Saarlandes, Saarbücken, Germany (July). http://www.ps.uni-saarland.de/alice/ Archived at Internet Archive: https://web.archive.org/web/20200201104850/http://www.ps.uni-saarland.de/alice/ (1 Feb. 2020 10:48:50)

Andrew Appel, David MacQueen, Robin Milner, and Mads Tofte. 1988. *Unifying Exceptions with Constructors in Standard ML.* Technical Report ECS-LFCS-88-55. LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, UK (May).

Andrew Appel and David B. MacQueen. 1991. Standard ML of New Jersey. In *Programming Language Implementation and Logic Programming (PLILP '91)* (*Lecture Notes in Computer Science*), J. Maluszynski and M. Wirsing (Eds.), Vol. 528. Springer-Verlag, New York, NY, USA, 1–13. https://doi.org/10.1007/3-540-54444-5_83

Andrew W. Appel. 1987. Garbage collection can be faster than stack allocation. *Inform. Process. Lett.* 25, 4 (June), 275–279. https://doi.org/10.1016/0020-0190(87)90175-X

Andrew W. Appel. 1989a. Runtime Tags Aren't Necessary. *Journal of Lisp and Symbolic Computation* 2, 2 (June), 153–62. https://doi.org/10.1007/BF01811537

Andrew W. Appel. 1989b. Simple Generational Garbage Collection and Fast Allocation. *Software – Practice and Experience* 19, 2, 171–183. https://doi.org/10.1002/spe.4380190206

Andrew W. Appel. 1990. A Runtime System. *Journal of Lisp and Symbolic Computation* 3, 4 (Nov.), 343–380. https://doi.org/10.1007/BF01807697

Andrew W. Appel. 1992. *Compiling with Continuations.* Cambridge University Press, Cambridge, UK.

Andrew W. Appel. 1993. A critique of Standard ML. *Journal of Functional Programming* 3, 4 (Oct.), 391–429. https://doi.org/10.1017/S0956796800000836

Andrew W. Appel. 1994. Proposed interface for Standard ML Stream I/O. Nov. 1994. 24 pages. http://sml-family.org/history/io.94-11-14.pdf Archived at Internet Archive: https://web.archive.org/web/20200314181846/http://sml-family.org/history/io.94-11-14.pdf (14 March 2020 18:18:46)

Andrew W. Appel. 1995. Proposed interface for Standard ML Stream I/O. July 1995. 24 pages. http://sml-family.org/history/io.95-07-13.pdf Archived at Internet Archive: https://web.archive.org/web/20200315172956/http://sml-family.org/history/io.95-07-13.pdf (15 March 2020 17:29:56)

Andrew W. Appel. 1998. *Modern Compiler Implementation in ML.* Cambridge University Press, Cambridge, UK.

Andrew W. Appel, Matthew Arcus, Nick Barnes, Dave Berry, Richard Brooksby, Emden R. Gansner, Lal George, Lorenz Huelsbergen, Dave MacQueen, Brian Monahan, John Reppy, Jon Thackray, and Peter Sestoft. 1995. A New Initial Basis

for Standard ML (Draft). June 1995. 113 pages. http://sml-family.org/history/basis-1995-06-26.pdf Archived at Internet Archive: https://web.archive.org/web/20200314182655/http://sml-family.org/history/basis-1995-06-26.pdf (14 March 2020 18:26:55) Draft design document dated June 26, 1995.

Andrew W. Appel, Dave Berry, Emden R. Gansner, Lal George, Lorenz Huelsbergen, Dave MacQueen, and John Reppy. 1994. A New Initial Basis for Standard ML (Draft). March 1994. 74 pages. http://sml-family.org/history/basis-1994-03-05.pdf Archived at Internet Archive: https://web.archive.org/web/20200313180417/http://sml-family.org/history/basis-1994-03-05.pdf (13 March 2020 18:04:17) Draft design document dated March 5, 1994.

Andrew W. Appel and Marcelo J. R. Concalves. 1993. *Hash-consing Garbage Collection.* Technical Report CS-TR-412-93. Department of Computer Science, Princeton University, Princeton, NJ, USA (Feb.).

Andrew W. Appel and Lal George. 2001. Optimal Spilling for CISC Machines with Few Registers. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '01)* (Snowbird, UT, USA). Association for Computing Machinery, New York, NY, USA (June), 243–253. https://doi.org/10.1145/378795.378854

Andrew W. Appel and Trevor Jim. 1989. Continuation-Passing, Closure-Passing Style. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages (POPL '89)* (Austin, TX, USA). Association for Computing Machinery, New York, NY, USA, 293–302. https://doi.org/10.1145/75277.75303

Andrew W. Appel and David B. MacQueen. 1987. A Standard ML Compiler. In *Functional Programming Languages and Computer Architecture (FPCA '87)* (Portland, OR, USA) (*Lecture Notes in Computer Science*), Vol. 274. Springer-Verlag, New York, NY, USA (Sept.), 301–324. https://doi.org/10.1007/3-540-18317-5_17

Andrew W. Appel and David B. MacQueen. 1994. Separate Compilation for Standard ML. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '94).* Association for Computing Machinery, New York, NY, USA (June), 13–23. https://doi.org/10.1145/178243.178245

Andrew W. Appel and Zhong Shao. 1992. Callee-save Registers in Continuation-Passing Style. *Journal of Lisp and Symbolic Computation* 5 (Sept.), 191–221. https://doi.org/10.1007/BF01807505

David Aspinall. 1995. Subtyping with singleton types. In *Proceedings of the 8th Workshop on Computer Science Logic (CSL '94)* (Kazimierz, Poland) (*Lecture Notes in Computer Science*). Springer-Verlag, New York, NY, USA (Sept.), 1–15. https://doi.org/10.1007/BFb0022243

Troy Kaighin Astarte. 2019. *Formalising Meaning, a History of Programming Language Semantics.* Ph.D. Dissertation. School of Computing, Newcastle University (June). Advisor: Cliff Jones.

Sven Auhagen, Lars Bergstrom, Matthew Fluet, and John Reppy. 2011. Garbage Collection for Multicore NUMA Machines. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC 2011).* Association for Computing Machinery, New York, NY, USA (June), 51–57. https://doi.org/10.1145/1988915.1988929

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering Formal Metatheory. In *Conference Record of the 35th Annual ACM Symposium on Principles of Programming Languages (POPL '08)* (San Francisco, CA, USA). Association for Computing Machinery, New York, NY, USA (Jan.), 3–15. https://doi.org/10.1145/1328438.1328443

H. P. Barendregt. 1992. Lambda calculi with types. In *Handbook of Logic in Computer Science (Volume 2).* Oxford University Press, Oxford, UK, 117–309.

D. W. Barron, J. N. Buxton, D. F. Hartley, E. Nixon, and C. Strachey. 1963. The Main Features of CPL. *Comput. J.* 6, 2 (Aug.), 134–143. https://doi.org/10.1093/comjnl/6.2.134

Marianne Baudinet and David MacQueen. 1985. Tree Pattern Matching for ML (extended abstract). Dec. 1985. http://sml-family.org/history/Baudinet-DM-tree-pat-match-12-85.pdf Archived at Internet Archive: https://web.archive.org/web/20200316180154/http://sml-family.org/history/Baudinet-DM-tree-pat-match-12-85.pdf (16 March 2020 18:01:54)

Mike Beaven and Ryan Stansifer. 1993. Explaining Type Errors in Polymorphic Languages. *ACM Letters on Programming Languages and Systems* 2, 1–4 (March–December), 17–30. https://doi.org/10.1145/176454.176460

Nick Benton and Andrew Kennedy. 1999. Monads, Effects and Transformations. *Electronic Notes in Theoretical Computer Science* 26, 3 – 20. https://doi.org/10.1016/S1571-0661(05)80280-4 Higher Order Operational Techniques in Semantics (HOOTS '99).

Nick Benton and Andrew Kennedy. 2001. Exceptional syntax. *Journal of Functional Programming* 11, 4, 295–410. https://doi.org/10.1017/S0956796801004099

Nick Benton, Andrew Kennedy, Sam Lindley, and Claudio Russo. 2005. Shrinking Reductions in SML.NET. In *Implementation and Application of Functional Languages*, Clemens Grelck, Frank Huch, Greg J. Michaelson, and Phil Trinder (Eds.). Springer-Verlag, New York, NY, USA, 142–159. https://doi.org/10.1007/11431664_9

Nick Benton, Andrew Kennedy, and George Russell. 1998. Compiling Standard ML to Java Bytecodes. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)* (Baltimore, MD, USA). Association for Computing Machinery, New York, NY, USA (Sept.), 129–140. https://doi.org/10.1145/289423.289435

Nick Benton, Andrew Kennedy, and Claudio V. Russo. 2004. Adventures in interoperability: the SML.NET experience. In *Proceedings of the 6th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP*

'04) (Verona, Italy). Association for Computing Machinery, New York, NY, USA (Aug.), 215–226. https://doi.org/10.1145/1013963.1013987

Karen L. Bernstein and Eugene W. Stark. 1995. *Debugging Type Errors (Full Version).* Technical Report. State University of New York and Stony Brook. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.36.3711 Archived at Internet Archive: https://web.archive.org/web/20130515040737/http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.36.3711 (15 May 2013 04:07:37)

Edoardo Biagioni, Robert Harper, Peter Lee, and Brian G. Milnes. 1994. Signatures for a Network Protocol Stack: A Systems Application of Standard ML. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming (LFP '94)* (Orlando, FL, USA). Association for Computing Machinery, New York, NY, USA (June), 55–64. https://doi.org/10.1145/182590.182431

Lars Birkedal, Nick Rothwell, Mads Tofte, and David N. Turner. 1993. *The ML Kit (Version 1).* Technical Report 93/14. Department of Computer Science, University of Copenhagen (March).

Matthias Blume. 2001. No-Longer-Foreign: Teaching an ML compiler to speak C "natively.". In *First workshop on multi-language infrastructure and interoperability (BABEL '01)* (Firenze, Italy) (*Electronic Notes in Theoretical Computer Science*), Vol. 59. Elsevier, New York, NY, USA (Sept.), 16. Issue 1. https://doi.org/10.1016/S1571-0661(05)80452-9

Matthias Blume, Umut A. Acar, and Wonseok Chae. 2006. Extensible Programming with First-Class Cases. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP '06)* (Portland, OR, USA). Association for Computing Machinery, New York, NY, USA (Sept.), 239–250. https://doi.org/10.1145/1159803.1159836

Matthias Blume, Umut A. Acar, and Wonseok Chae. 2008. Exception Handlers as Extensible Cases. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems (APLAS '08)* (Bangalore, India) (*Lecture Notes in Computer Science*), G. Ramalingam (Ed.), Vol. 5356. Springer-Verlag, New York, NY, USA (Dec.), 273–289. https://doi.org/10.1007/978-3-540-89330-1_20

Matthias Blume and Andrew W. Appel. 1999. Hierarchical Modularity. *ACM Transactions on Programming Languages and Systems* 21, 4 (July), 813–847. https://doi.org/10.1145/325478.325518

Mark R. Brown and Greg Nelson. 1989. *IO Streams: Abstract Types, Real Programs.* Technical Report 53. Digital Systems Research Center, Palo Alto, CA, USA (Nov.).

R. Burstall and J. Goguen. 1977. Putting theories together to make specifications. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence (II)* (Cambridge, MA, USA), Raj Reddy (Ed.). AAAI Press, Palo Alto, CA, USA (Aug.), 1045–1058.

Rod M. Burstall. 1977. Design Considerations for a Functional Programming Language. In *The Software Revolution: Proceedings of the Infotech State of the Art Conference* (Copenhagen, Denmark). Infotech and Pergamon Press, Oxford, UK (Oct.), 45–57.

Rod M. Burstall, J. S. Collins, and R. J. Popplestone. 1977. *Programming in POP-2.* Edinburgh University Press, Edinburgh, UK.

Rod M. Burstall and John Darlington. 1977. A Tranformation System for Developing Recursive Programs. *Journal of the ACM* 24, 1 (Jan.), 44–67. https://doi.org/10.1145/321992.321996

Rod M. Burstall and Peter Landin. 1969. Programs and their Proofs: an Algebraic Approach. In *Machine Intelligence*, B. Meltzer and D. Michie (Eds.). Vol. 4. Elsevier, New York, NY, USA, 17–43.

Rod M. Burstall, David B. MacQueen, and Donald Sannella. 1980. HOPE: an experimental applicative language. In *Conference Record of the 1980 ACM Conference on Lisp and Functional Programming (LFP '80).* Association for Computing Machinery, New York, NY, USA (Aug.), 136–143. https://doi.org/10.1145/800087.802799

Rod M. Burstall and R. J. Popplestone. 1968. POP-2 reference manual. In *Machine Intelligence 2*, E. Dale and D. Mitchie (Eds.). Edinburgh University Press, Edinburgh, UK, 207–46.

Georg Cantor. 1874. Über eine Eigenschaft des Inbegriffes aller reellen algebraischen Zahlen. *Journal für die Reine und Angewandte Mathematik* 77, 258–262.

Luca Cardelli. 1980a. The ML Abstract Machine. Nov. 1980. http://sml-family.org/history/Cardelli-ML-abstract-machine-1980.pdf Archived at Internet Archive: https://web.archive.org/web/20200316/http://sml-family.org/history/Cardelli-ML-abstract-machine-1980.pdf (16 March 2020) Early description of the ML abstract machine (AM), precursor to the FAM.

Luca Cardelli. 1980b. A Module Exchange Format. Dec. 1980. http://sml-family.org/history/Cardelli-MEX-1980_12.pdf Archived at Internet Archive: https://web.archive.org/web/20200316202334/http://sml-family.org/history/Cardelli-MEX-1980_12.pdf (16 March 2020 20:23:34) Description of a textual format for exporting internal ML data structures.

Luca Cardelli. 1981. Differences Between VAX and DEC-10 ML. March 1981. 8 pages. http://sml-family.org/history/Cardelli-mlchanges_doc-1982_03.pdf Archived at Internet Archive: https://web.archive.org/web/20200317193405/http://sml-family.org/history/Cardelli-mlchanges_doc-1982_03.pdf (17 March 2020 19:34:05)

Luca Cardelli. 1982a. *An Algebraic Approach to Hardware Description and Verification.* Ph.D. Dissertation. University of Edinburgh (April).

Luca Cardelli. 1982b. Edinburgh ML. March 1982. 2 pages. http://sml-family.org/history/Cardelli-Edinburgh-ML-README-1982_03.pdf Archived at Internet Archive: https://web.archive.org/web/20200317194022/http://sml-family.org/history/Cardelli-Edinburgh-ML-README-1982_03.pdf (17 March 2020 19:40:22) README file for VAX ML (ML under VMS) distribution.

Luca Cardelli. 1982c. Known VAX-ML System Locations. *Polymorphism: The ML/LCF/Hope Newsletter* I, 0 (Nov.), 4–5. http://lucacardelli.name/Papers/Polymorphism%20Vol%20I,%20No%200.pdf Archived at Internet Archive: https://web.archive.org/web/20190225153220/http://lucacardelli.name/Papers/Polymorphism%20Vol%20I,%20No%200.pdf (25 Feb. 2019 15:32:20)

Luca Cardelli. 1982d. *ML under VMS*. Department of Computer Science, University of Edinburgh. http://sml-family.org/history/Cardelli-ML-VMS-manual-1982.pdf Archived at Internet Archive: https://web.archive.org/web/20200317193823/http://sml-family.org/history/Cardelli-ML-VMS-manual-1982.pdf (17 March 2020 19:38:23) Manual for ML under VMS.

Luca Cardelli. 1983a. The Functional Abstract Machine. *Polymorphism: The ML/LCF/Hope Newsletter* I, 1 (Jan.), 16–52. http://lucacardelli.name/Papers/Polymorphism%20Vol%20I,%20No%201.pdf Archived at Internet Archive: https://web.archive.org/web/20190307022704/http://lucacardelli.name/Papers/Polymorphism%20Vol%20I,%20No%201.pdf (7 March 2019 02:27:04)

Luca Cardelli. 1983b. *ML under Unix*. Bell Laboratories (Aug.). Manual for ML under Unix, Pose 2.

Luca Cardelli. 1983c. Stream Input/Output. *Polymorphism: The ML/LCF/Hope Newsletter* I, 3 (Dec.), 9. http://lucacardelli.name/Papers/Polymorphism%20Vol%20I,%20No%203.pdf Archived at Internet Archive: https://web.archive.org/web/20190307145556/http://lucacardelli.name/Papers/Polymorphism%20Vol%20I,%20No%203.pdf (7 March 2019 14:55:56)

Luca Cardelli. 1984a. Compiling a Functional Language. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming (LFP '84)* (Austin, TX, USA). Association for Computing Machinery, New York, NY, USA (Aug.), 208–217. https://doi.org/10.1145/800055.802037

Luca Cardelli. 1984b. *ML under Unix*. Bell Laboratories (April). Manual for ML under Unix, Pose 4.

Luca Cardelli and Xavier Leroy. 1990. Abstract Types and the Dot Notation. In *Proceedings of the IFIP TC2 Working Conference on Programming Concepts and Methods*. North Holland, Amsterdam, Netherlands (April), 479–504. also DEC SRC Tech Report 56.

Felice Cardone, Jr. and Roger Hindley. 2009. Lambda-Calculus and Combinators in the 20th Century. In *Logic from Russell to Church*. Handbook of the History of Logic, Vol. 5. Elsevier, New York, NY, USA, 723–817. Also available as Swansea University Mathematics Department Research Report No. MRRS-05-06.

Rudolf Carnap. 1929. *Abriss der Logistik, mit besonderer berücksichtigung der relationstheorie und ihrer anwendungen*. Springer-Verlag, Springer-Verlag. https://archive.org/details/RudolfCarnapAbrissDerLogistik

Henry Cejtin, Matthew Fluet, Suresh Jagannathan, and Stephen Weeks. 2004. Formal Specification of the ML Basis System. . 8 pages. http://mlton.org/MLBasis.attachments/mlb-formal.pdf Archived at Internet Archive: https://web.archive.org/web/20160909024016/http://mlton.org/MLBasis.attachments/mlb-formal.pdf (9 Sept. 2016 02:40:16)

Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. 2000. Flow-Directed Closure Conversion for Typed Languages. In *Proceedings of the 9th European Symposium on Programming Languages and Systems (ESOP '00)*. Springer-Verlag, New York, NY, USA (March–April), 56–71. https://doi.org/10.1007/3-540-46425-5_4

Perry Cheng and Guy E. Blelloch. 2001. A Parallel, Real-Time Garbage Collector. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '01)* (Snowbird, UT, USA). Association for Computing Machinery, New York, NY, USA (June), 125–136. https://doi.org/10.1145/378795.378823

Olaf Chitil. 2001. Compositional Explanation of Types and Algorithmic Debugging of Type Errors. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01)* (Florence, Italy). Association for Computing Machinery, New York, NY, USA (Sept.), 193–204. https://doi.org/10.1145/507635.507659

Charisee Chiw, Gordon Kindlmann, John Reppy, Lamont Samuels, and Nick Seltzer. 2012. Diderot: A Parallel DSL for Image Analysis and Visualization. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)* (Beijing, China). Association for Computing Machinery, New York, NY, USA (June), 111–120. https://doi.org/10.1145/2254064.2254079

Adam Chlipala. 2015. Ur/Web: A Simple Model for Programming the Web. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India). Association for Computing Machinery, New York, NY, USA (Jan.), 153–165. https://doi.org/10.1145/2676726.2677004

Venkatesh Choppella. 2002. *Unification Source-Tracking with Application to Diagnosis of Type Inference*. Ph.D. Dissertation. Department of Computer Science, University of Indiana, Bloomington, IN, USA (Aug.).

Alonzo Church. 1940. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic* 5, 2 (June), 56–68.

Rance Cleaveland, Joachim Parrow, and Bernard Steffen. 1993. The Concurrency Workbench: A Semantics-based Tool for the Verification of Concurrent Systems. *ACM Transactions on Programming Languages and Systems* 15, 1 (Jan.), 36–72. https://doi.org/10.1145/151646.151648

P. M. Cohn. 1965. *Universal Algebra.* Harper Row, New York and London.

Robert Cooper and Clifford D. Krumvieda. 1992. Distributed Programming with Asynchronous Ordered Channels in Distributed ML. In *Proceedings of the 1992 ACM SIGPLAN Workshop on ML and its Applications.* Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA (June), 134–148. Proceedings available as Technical Report CMU-CS-93-105.

Thierry Coquand. 2018. Type Theory. In *The Stanford Encyclopedia of Philosophy (Fall 2018 Edition)*, Edward N. Zalta (Ed.). The Metaphysics Research Lab, Center for the Study of Language and Information, Stanford University, Stanford, CA, USA (July). https://plato.stanford.edu/entries/type-theory/ Archived at Internet Archive: https://web.archive.org/web/20191211203338/https://plato.stanford.edu/entries/type-theory/ (11 Dec. 2019 20:33:38)

Karl Crary and Robert Harper. 2009. *Mechanization of Type Safety of Standard ML.* Carnegie Mellon University (Aug.). https://doi.org/10.5281/zenodo.3700588 Archived at Zenodo: https://zenodo.org/record/3700588/files/mechdefsml.tar.gz

Karl Crary, Robert Harper, and Sidd Puri. 1999. What is a Recursive Module?. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99)* (Atlanta, GA, USA). Association for Computing Machinery, New York, NY, USA (May), 50–63. https://doi.org/10.1145/301631.301641

Pierre Crégut and David B. MacQueen. 1994. An Implementation of Higher-Order Functors. In *Proceedings of the 1994 ACM SIGPLAN Workshop on ML and its Applications* (Orlando, FL, USA). INRIA, Rocquencourt, France (June), 13–22.

H. B. Curry. 1930. Grundlagen der kombinatorischen Logik. *American Journal of Mathematics* 52, 509–536, 789–834.

H. B. Curry. 1932. Some additions to the theory of combinators. *American Journal of Mathematics* 54, 551–558.

H. B. Curry. 1934. Functionality in Combinatory Logic. *Proceedings of the National Academy of Sciences of the United States of America* 20, 11 (Nov.), 584–590.

H. B. Curry. 1969. Modified basic functionality in combinatory logic. *Dialectica* 23, 83–92.

H. B. Curry and R. Feys. 1958. *Combinatory Logic, Volume I.* North Holland, Amsterdam, Netherlands. 3rd edition 1974.

Luis Damas. 1984. *Type Assignment in Programming Languages.* Ph.D. Dissertation. Department of Computer Science, University of Edinburgh, Edignburgh, UK.

Luis Damas and Robin Milner. 1982. Principal type-schemes for functional programs. In *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages (POPL '82).* Association for Computing Machinery, New York, NY, USA (Jan.), 207–212. https://doi.org/10.1145/582153.582176

Allyn Dimock, Ian Westmacott, Robert Muller, Franklyn Turbak, and J. B. Wells. 2001. Functioning without Closure: Type-Safe Customized Function Representations for Standard ML. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01)* (Florence, Italy). Association for Computing Machinery, New York, NY, USA (Sept.), 14–25. https://doi.org/10.1145/507669.507640

Derek Dreyer. 2005. *Understanding and Evolving the ML Module System.* Ph.D. Dissertation. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA (May).

Derek Dreyer. 2007. A Type System for Recursive Modules. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP '07).* Association for Computing Machinery, New York, NY, USA (Oct.), 289–302. https://doi.org/10.1145/1291220.1291196

Derek Dreyer, Karl Crary, and Robert Harper. 2003. A Type System for Higher-Order Modules. In *Conference Record of the 30th Annual ACM Symposium on Principles of Programming Languages (POPL '03).* Association for Computing Machinery, New York, NY, USA (Jan.), 236–249. https://doi.org/10.1145/640128.604151

Derek Dreyer, Robert Harper, Manuel M. T. Chakravarty, and Gabrielle Keller. 2007. Modular type classes. In *Conference Record of the 34th Annual ACM Symposium on Principles of Programming Languages (POPL '07).* Association for Computing Machinery, New York, NY, USA (Jan.), 63–70. https://doi.org/10.1145/1190215.1190229

Derek Dreyer and Andreas Rossberg. 2008. Mixin' up the ML Module System. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)* (Victoria, BC, Canada). Association for Computing Machinery, New York, NY, USA (Sept.), 307–320. https://doi.org/10.1145/2450136.2450137

Derek R. Dreyer, Robert Harper, and Karl Crary. 2001. *Toward a Practical Type Theory for Recursive Modules.* Technical Report CMU-CS-01-112. School of Computer Science, Carnegie Mellon University (March).

Dominic Duggan and Frederick Bent. 1996. Explaining type inference. *Science of Computer Programming* 27, 1 (July), 37–83.

Dominic Duggan and Constantinos Sourelis. 1996. Mixin Modules. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP '96)* (Philadelphia, PA, USA). Association for Computing Machinery, New York, NY, USA (May), 262–273. https://doi.org/10.1145/232627.232654

H. Ehrig, H.-J. Kreowski, J. Thatcher, E. Wagner, and J. Wright. 1980. Parameterized data types in algebraic specification languages. In *Automata, Languages and Programming (ICALP '80)* (Noordwijkerhout, Netherlands) (*Lecture Notes in Computer Science*), J. de Bakker and J. van Leeuwen (Eds.), Vol. 85. Springer-Verlag, New York, NY, USA (July), 157–168. https://doi.org/10.1007/3-540-10003-2_68

Peter Harry Eidorff, Fritz Henglein, Christian Mossin, Henning Niss, Morten Heine Sørensen, and Mads Tofte. 1999. AnnoDomini: From Type Theory to Year 2000 Conversion Tool. In *Conference Record of the 26th Annual ACM Symposium*

on Principles of Programming Languages (POPL '99) (San Antonio, TX, USA). Association for Computing Machinery, New York, NY, USA (Jan.), 1–14. https://doi.org/10.1145/292540.292543

Martin Elsman. 1998. Polymorphic Equality — No Tags Required. In *Proceedings of the Second International Workshop on Types in Compilation* (*Lecture Notes in Computer Science*). Springer-Verlag, New York, NY, USA (March), 136–155. https://doi.org/10.1007/BFb0055516

Martin Elsman. 1999a. *Program Modules, Separate Compilation, and Intermodule Optimisation.* Ph.D. Dissertation. Department of Computer Science, University of Copenhagen (Jan.).

Martin Elsman. 1999b. Static Interpretation of Modules. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99).* Association for Computing Machinery, New York, NY, USA (Sept.), 208–219. https://doi.org/10.1145/317765.317800

Martin Elsman. 2011. SMLtoJs: Hosting a Standard ML Compiler in a Web Browser. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients (PLASTIC '11)* (Portland, OR, USA). Association for Computing Machinery, New York, NY, USA, 39–48. https://doi.org/10.1145/2093328.2093336

Martin Elsman, Jeffrey S. Foster, and Alexander Aiken. 1999. *Carillon — A System to Find Y2K Problems in C Programs.* Computer Science Division, University of California, Berkeley (July). User's Manual.

Martin Elsman and Niels Hallenberg. 2003. Web Programming with SMLserver. In *Fifth International Symposium on Practical Aspects of Declarative Languages (PADL'03)* (New Orleans, LA, USA) (*Lecture Notes in Computer Science*). Springer-Verlag, New York, NY, USA (Jan.), 74–91. https://doi.org/10.1007/3-540-36388-2_7

Arthur Evans, Jr. 1968a. PAL – a language designed for teaching programming linguistics. In *Proceedings of the 1968 23rd ACM National Conference (ACM '68).* Association for Computing Machinery, New York, NY, USA (Aug.), 395–403. https://doi.org/10.1145/800186.810604

Arthur Evans, Jr. 1968b. *PAL – A reference manual and a primer.* Technical Report. MIT Department of Electrical Engineering.

Arthur Evans, Jr. 1970. *PAL – Pedagigic Algorithmic Language.* Technical Report. MIT Department of Electrical Engineering (Feb.).

Matthias Felleisen and Daniel P. Friedman. 1998. *The Little MLer.* The MIT Press, Cambridge, MA, USA.

Kathleen Fisher and John Reppy. 1999. The design of a class mechanism for Moby. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99)* (Atlanta, GA, USA). Association for Computing Machinery, New York, NY, USA (May), 37–49. https://doi.org/10.1145/301618.301638

Kathleen Fisher and John Reppy. 2002. Inhertance-based subtyping. *Information and Computation* 177, 1 (Aug.), 28–55. https://doi.org/10.1006/inco.2002.3169

Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. 2010. Implicitly Threaded Parallelism in Manticore. *Journal of Functional Programming* 20, 5-6 (Nov.), 537–576. https://doi.org/10.1017/S0956796810000201

Gottlob Frege. 1879. *Begriffsschrift.* Verlag Louis Nebert, Halle, Germany. Also available in English as Begriffsschrift in van Heijenoort, 1967, pp. 1–82.

Gottlob Frege. 1893. *Grundgesetze der Arithmetik.* Verlag Hermann Pohle, Jena, Germany.

Emden R. Gansner and John H. Reppy. 1991. eXene. In *Third International Workshop on Standard ML.* Carnegie Mellon University, Pittsburgh, PA, USA (Sept.), 16. https://people.cs.uchicago.edu/~jhr/papers/1991/ml-exene.pdf Archived at Internet Archive: https://web.archive.org/web/20200328152609/https://people.cs.uchicago.edu/~jhr/papers/1991/ml-exene.pdf (28 March 2020 15:26:09)

Emden R. Gansner and John H. Reppy. 1993. A Multi-threaded Higher-order User Interface Toolkit. In *User Interface Software, Bass and Dewan (Eds.).* Software Trends, Vol. 1. John Wiley & Sons, Inc., New York, NY, USA, 61–80.

Emden R. Gansner and John H. Reppy (Eds.). 2004. *The Standard ML Basis Library.* Cambridge University Press, Cambridge, UK.

Lal George and Andrew W. Appel. 1996. Iterated Register Coalescing. *ACM Transactions on Programming Languages and Systems* 18, 3 (May), 300–324. https://doi.org/10.1145/229542.229546

Lal George and Matthias Blume. 2003. Taming the IXP Network Processor. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '03)* (San Diego, CA, USA). Association for Computing Machinery, New York, NY, USA (June), 26–37. https://doi.org/10.1145/781131.781135

Lal George, Florent Guillame, and John H. Reppy. 1994. A Portable and Optimizing Back End for the SML/NJ Compiler. In *Proceedings of the 5th International Conference on Compiler Construction (CC '94).* Springer-Verlag, New York, NY, USA (April), 83–97. https://doi.org/10.1007/3-540-57877-3_6

Stephen Gilmore. 1997. *Programming in Standard ML '97: A Tutorial Introduction.* Technical Report ECS-LFCS-97-364. LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, UK (Sept.). Also available online at http://homepages.inf.ed.ac.uk/stg/NOTES.

Jean-Yves Girard. 1972. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Ph.D. Dissertation. Université Paris VII. Thése d'état.

Benjamin Goldberg. 1991. Tag-free Garbage Collection for Strongly Typed Programming Languages. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '91)* (Toronto, Canada). Association for Computing Machinery, New York, NY, USA (June), 165–176. https://doi.org/10.1145/113446.113460

Benjamin Goldberg and Michael Gloger. 1992. Polymorphic Type Reconstruction for Garbage Collection Without Tags. In *Conference record of the 1992 ACM Conference on Lisp and Functional Programming (LFP '92)* (San Francisco, CA, USA). Association for Computing Machinery, New York, NY, USA (June), 53–65. https://doi.org/10.1145/141471.141504

M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth. 1978a. A Metalanguage for Interactive Proof in LCF. In *Conference Record of the 5th Annual ACM Symposium on Principles of Programming Languages (POPL '78)*. Association for Computing Machinery, New York, NY, USA (Jan.), 119–130. https://doi.org/10.1145/512760.512773

Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. 1979. *Edinburgh LCF*. Lecture Notes in Computer Science, Vol. 78. Springer-Verlag, New York, NY, USA. https://doi.org/10.1007/3-540-09724-4

Michael J. C. Gordon. 1980. Locations as first class objects in ML. . http://sml-family.org/history/Gordon-ML-refs-1980.pdf Archived at Internet Archive: https://web.archive.org/web/20200317194215/http://sml-family.org/history/Gordon-ML-refs-1980.pdf (17 March 2020 19:42:15) Note to Luis Damas proposing a research topic.

M. J. C. Gordon, R. Milner, L. Morris, M. C. Newey, and C. P. Wadsworth. 1978b. A metalanguage for interactive proof in LCF. In *Conference Record of the 5th Annual ACM Symposium on Principles of Programming Languages (POPL '78)*. Association for Computing Machinery, New York, NY, USA (Jan.), 119–130. https://doi.org/10.1145/512760.512773

Timoth Griffin and Howard Trickey. 1994. Integrity Maintenance in a Telecommunications Switch. *Bulletin of the Technical Committee on Data Engineering* 17, 2 (June), 43–46.

Christian Haack and J. B. Wells. 2004. Type Error Slicing in Implicitly Typed Higher-Order Languages. *Science of Computer Programming* 50, 1–3 (March), 189–224. https://doi.org/10.1016/j.scico.2004.01.004

Jurriaan Hage and Bastiaan Heeren. 2006. Heuristics for type error discovery and recovery. In *18th International Symposium on the Implementation and Application of Functional Languages (IFL '06)* (Budapest, Hungary) (*Lecture Notes in Computer Science*). Springer-Verlag, New York, NY, USA, 199–216. https://doi.org/10.1007/978-3-540-74130-5_12

Nick Haines. 1991. Compiling ML to a Strongly-typed System. In *Third International Workshop on Standard ML*. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA (Sept.), 11.

Niels Hallenberg, Martin Elsman, and Mads Tofte. 2002. Combining Region Inference and Garbage Collection. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '02)* (Berlin, Germany). Association for Computing Machinery, New York, NY, USA (June), 141–152. https://doi.org/10.1145/512529.512547

Kevin Hammond. 1988. *Parallel SML: a Functional Language and its Implementation in Dactl*. Pitman, London, UK.

Wilfred J. Hansen. 1992. Subsequence References: First-class Values for Substrings. *ACM Transactions on Programming Languages and Systems* 14, 4 (Oct.), 471–489. https://doi.org/10.1145/133233.133234

D. R. Hanson. 1990. Fast Allocation and Deallocation of Memory Based on Object Lifetimes. *Software – Practice and Experience* 20, 1 (Jan.), 5–12. https://doi.org/10.1002/spe.4380200104

Robert Harper. 1985. Report on the Standard ML Meeting, Edinburgh, May 23–25, 1985 (DRAFT). . 10 pages. http://sml-family.org/history/Harper-SML-meeting-1985_05.pdf Archived at Internet Archive: https://web.archive.org/web/20200316201215/http://sml-family.org/history/Harper-SML-meeting-1985_05.pdf (16 March 2020 20:12:15)

Robert Harper. 1986. Standard ML Input/Output. In *Standard ML*. Number ECS-LFCS-86-2 in LFCS Report Series. LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, UK (March). This paper is dated June 10, 1985.

Robert Harper. 1994. A Simplified Account of Polymorphic References. *Inform. Process. Lett.* 51, 201–206. https://doi.org/10.1016/0020-0190(94)90120-1

Robert Harper. 2011. *Programming in Standard ML*. Carnegie Mellon University (Feb.). http://www.cs.cmu.edu/~rwh/isml/book.pdf Archived at Internet Archive: https://web.archive.org/web/20191212212327/http://www.cs.cmu.edu/~rwh/isml/book.pdf (12 Dec. 2019 21:23:27)

Robert Harper. 2016. *Practical Foundations for Programming Languages* (2 ed.). Cambridge University Press, Cambridge, UK.

Robert Harper, Furio Honsell, and Gordon Plotkin. 1993a. A Framework for Defining Logics. *Journal of the ACM* 40, 1 (Jan.), 143–184. https://doi.org/10.1145/138027.138060

Robert Harper and Peter Lee. 1994. *Advanced Languages for System Software — The Fox Project in 1994*. Technical Report CMU-CS-94-104. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA (Jan.).

Robert Harper, Peter Lee, Frank Pfenning, and Gene Rollins. 1994. A Compilation Manager for SML/NJ. In *Proceedings of the 1994 ACM SIGPLAN Workshop on ML and its Applications* (Orlando, FL, USA) (*INRIA Technical Report*). INRIA, Rocquencourt, France (June), 136–147.

Robert Harper and Mark Lillibridge. 1994. A type-theoretic approach to higher-order modules with sharing. In *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages (POPL '94)*. Association for Computing Machinery, New York, NY, USA (Jan.), 123–137. https://doi.org/10.1145/174675.176927

Robert Harper, David MacQueen, and Robin Milner. 1986. *Standard ML*. Technical Report ECS-LFCS-86-2. LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, UK (March).

Robert Harper, David B. MacQueen, and Bruce F. Duba. 1993b. Typing first-class continuations in ML. *Journal of Functional Programming* 3 (Oct.), 465–484. 10.1017/S095679680000085X

Robert Harper, Robin Milner, and Mads Tofte. 1987a. *The Semantics of Standard ML, Version 1.* Technical Report ECS-LFCS-87-36. LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, UK (Aug.).

R. Harper, R. Milner, and M. Tofte. 1987b. A type discipline for program modules. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT '87)* (*Lecture Notes in Computer Science*), H. Ehrig, R. Kowalski, G. Levi, and U. Montanari (Eds.). Springer-Verlag, New York, NY, USA (March), 308–319. https://doi.org/10.1007/BFb0014988

Robert Harper, Robin Milner, and Mads Tofte. 1988. *The Definition of Standard ML, Version 2.* Technical Report ECS-LFCS-88-62. LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, UK (Aug.).

Robert Harper, Robin Milner, and Mads Tofte. 1989. *The Definition of Standard ML, Version 3.* Technical Report ECS-LFCS-89-81. LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, UK (May).

Robert Harper and John C. Mitchell. 1993. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems* 15, 2, 211–252. https://doi.org/10.1145/169701.169696

Robert Harper, John C. Mitchell, and Eugenio Moggi. 1990. Higher-order modules and the phase distinction. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages (POPL '90).* Association for Computing Machinery, New York, NY, USA (Jan.), 341–354. https://doi.org/10.1145/96709.96744

Robert Harper and Christopher Stone. 2000. A type-theoretic interpretation of Standard ML. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Gordon Plotkin, Colin Sterling, and Mads Tofte (Eds.). The MIT Press, Cambridge, MA, USA, 341–388.

D. Hilbert and W. Ackermann. 1928. *Grundzüge der Theoretischen Logik.* Springer-Verlag, Berlin, Germany.

J. Roger Hindley. 1969. The principal type scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.* 146 (Dec.), 29–60. https://doi.org/10.2307/1995158

J. Roger Hindley. 1997. *Basic Simple Type Theory.* Cambridge Tracts in Theoretical Computer Science, Vol. 42. Cambridge University Press, Cambridge, UK.

J. Roger Hindley. 2008. M. H. Newman's Typability Algorithm for Lambda-calculus. *Journal of Logic and Computation* 18, 2, 229–238. https://doi.org/10.1093/logcom/exm001

Tom Hirschowitz and Xavier Leroy. 2005. Mixin modules in a call-by-value setting. *ACM Transactions on Programming Languages and Systems* 27, 5 (Sept.), 857–881. https://doi.org/10.1145/1086642.1086644

HOL 2019. *HOL: Interactive Theorem Prover.* HOL (Aug.). https://hol-theorem-prover.org Archived at Internet Archive: https://web.archive.org/web/20190925051532/https://hol-theorem-prover.org/ (25 Sept. 2019 05:15:32)

S. Holmström. 1983. PFL: A functional language for parallel programming. In *Declarative Programming Workshop.* University College, London, UK (April), 114–139. Extended version published as Report 7, Programming Methodology Group, Chalmers University, September 1983.

IEEE. 1985. *IEEE Standard for Binary Floating-Point Arithmetic.* Technical Report IEEE Std 754-1985. IEEE (Aug.).

IEEE. 1993. *Portable Operating System Interface — Part 1: System Application Program Interface (API) [C Language].* Technical Report IEEE Std 1003.1. IEEE.

The University of Cambridge 2019. *Isabelle.* The University of Cambridge (June). https://www.cl.cam.ac.uk/research/hvg/Isabelle/index.html Archived at Internet Archive: https://web.archive.org/web/20190927025407/https://www.cl.cam.ac.uk/research/hvg/Isabelle/index.html (27 Sept. 2019 02:54:07)

Suresh Jagannathan, Armand Navabi, KC Sivaramakrishnan, and Lukasz Ziarek. 2010. The Design Rationale for Multi-MLton. In *Proceedings of the 2010 ACM SIGPLAN Workshop on ML (ML '10)* (Baltimore, MD, USA). Association for Computing Machinery, New York, NY, USA (Sept.), 2. https://multimlton.cs.purdue.edu/mML/Publications_files/MultiMLton-MLW.pdf Archived at Internet Archive: https://web.archive.org/web/20200328161027/https://multimlton.cs.purdue.edu/mML/Publications_files/MultiMLton-MLW.pdf (28 March 2020 16:10:27)

Kathleen Jensen and Niklaus Wirth. 1978. *Pascal User Manual and Report* (2nd ed.). Springer-Verlag, New York, NY, USA.

Gregory F. Johnson and Janet A. Walz. 1986. A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages (POPL '86).* Association for Computing Machinery, New York, NY, USA (Jan.), 44–57. https://doi.org/10.1145/512644.512649

Gilles Kahn. 1987. Natural Semantics. In *Symposium on Theoretical Aspects of Computer Science (STACS '87)* (*Lecture Notes in Computer Science*), Vol. 247. Springer-Verlag, New York, NY, USA, 22–39. https://doi.org/10.1007/BFb0039592

Stefan Kahrs. 1993. *Mistakes and ambiguities in the Definition of Standard ML.* Technical Report ECS-LFCS-93-257. LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, UK (April).

Stefan Kahrs and Donald Sannella. 1998. Reflections on the Design of a Specification language. In *Proceedings of the First International Conference on Fundamental Approaches to Software Engineering (FASE '98)* (*Lecture Notes in Computer Science*). Springer-Verlag, New York, NY, USA, 154–170. https://doi.org/10.1007/BFb0053589

Stefan Kahrs, Don Sannella, and Andrzej Tarlecki. 1994. *The Definition of Extended ML.* Technical Report ECS-LFCS-94-300. LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, UK.

Stefan Kahrs, Donald Sannella, and Andrzej Tarlecki. 1997. The Definition of Extended ML: A Gentle Introduction. *Theoretical Computer Science* 173, 2, 445–484. https://doi.org/10.1016/S0304-3975(96)00163-6

Fairouz Kamareddine, Twan Laan, and Rob Nederpelt. 2004. *A Modern Perspective on Type Theory: From its Origins until Today.* Applied Logic Series, Vol. 29. Kluwer Academic Publishers, Dordrecht/Boston/London.

Andrew Kennedy. 2007. Compiling with Continuations, Continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP '07)* (Freiburg, Germany). Association for Computing Machinery, New York, NY, USA (Oct.), 177–190. https://doi.org/10.1145/1291151.1291179

Andrew Kennedy and Don Syme. 2001. Design and Implementation of Generics for the .NET Common Language Runtime. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '01)* (Snowbird, UT, USA). Association for Computing Machinery, New York, NY, USA (June), 1–12. https://doi.org/10.1145/378795.378797

Andrew J. Kennedy. 1996. *Programming languages and dimensions.* Ph.D. Dissertation. University of Cambridge, Computer Science Department, Cambridge, UK (April). Also available as Technical Report CL-TR-391.

Clifford D. Krumvieda. 1993. *Distributed ML: Abstractions for Efficient and Fault-Tolerant Programming.* Ph.D. Dissertation. Department of Computer Science, Cornell University, Ithaca, NY, USA (Aug.). Available as Technical Report TR 93-1376.

George Kuan. 2010. *A True Higher-Order Module System.* Ph.D. Dissertation. Department of Computer Science, University of Chicago, Chicago, IL, USA.

George Kuan and David MacQueen. 2007. Efficient ML Type Inference Using Ranked Type Variables. In *Proceedings of the 2007 ACM SIGPLAN Workshop on ML (ML '07).* Association for Computing Machinery, New York, NY, USA (Oct.), 3–14. https://doi.org/10.1145/1292535.1292538

George Kuan and David MacQueen. 2010. Engineering Higher-Order Modules in SML/NJ. In *21st International Workshop on the Implementation and Application of Functional Languages (IFL '09)* (South Orange, NJ, USA) (*Lecture Notes in Computer Science*), M. T. Morazán and S. B. Scholz (Eds.), Vol. 6041. Springer-Verlag, New York, NY, USA (Sept.), 218–235. https://doi.org/10.1007/978-3-642-16478-1_13

Peter J. Landin. 1964. The mechanical evaluation of expressions. *Comput. J.* 6, 4, 308–320. https://doi.org/10.1093/comjnl/6.4.308

Peter J. Landin. 1965a. A correspondence between ALGOL 60 and Church's lambda-notation: Part I. *Commun. ACM* 8, 2 (Feb.), 89–101. https://doi.org/10.1145/363744.363749

Peter J. Landin. 1965b. A correspondence between ALGOL 60 and Church's lambda-notation: Part II. *Commun. ACM* 8, 3 (March), 158–165. https://doi.org/10.1145/363791.363804

Peter J. Landin. 1966a. A Lambda-Calculus Approach. In *Advances in Programming and Non-Numerical Computation*, L. Fox (Ed.). Pergamon Press, Oxford, UK, Chapter 5, 97–141. https://doi.org/10.1016/B978-0-08-011356-2.50008-2

Peter J. Landin. 1966b. The next 700 programming languages. *Commun. ACM* 9, 3, 157–66. https://doi.org/10.1145/365230.365257

Peter J. Landin. 1998. A Generalization of Jumps and Labels. *Higher-order and Symbolic Computation* 11, 2 (Dec.), 117–123. https://doi.org/10.1023/A:1010068630801

Daniel K. Lee, Karl Crary, and Robert Harper. 2007. Towards a mechanized metatheory of Standard ML. In *Conference Record of the 34th Annual ACM Symposium on Principles of Programming Languages (POPL '07)* (Nice, France). Association for Computing Machinery, New York, NY, USA (Jan.), 173–184. https://doi.org/10.1145/1190216.1190245

Oukseh Lee and Kwangkeun Yi. 1998. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems* 20, 4 (July), 707–723. https://doi.org/10.1145/291891.291892

Daan Leijen and Erik Meijer. 1999. Domain Specific Embedded Compilers. In *Proceedings of the 2nd Conference on Domain-Specific Languages (DSL '99)* (Austin, TX, USA). The USENIX Association, Berkeley, CA, USA (Oct.), 109–122. https://doi.org/10.1145/331963.331977

Benjamin Lerner, Matthew Flower, , Dan Grossman, and Craig Chambers. 2007. Searching for Type-Error Messages. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)* (San Diego, CA, USA). Association for Computing Machinery, New York, NY, USA (June), 425–434. https://doi.org/10.1145/1250734.1250783

Xavier Leroy. 1993. Polymorphism by name for references and continuations. In *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages (POPL '93).* Association for Computing Machinery, New York, NY, USA (Jan.), 220–231. https://doi.org/10.1145/158511.158632

Xavier Leroy. 1994. Manifest types, modules, and separate compilation. In *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages (POPL '94).* Association for Computing Machinery, New York, NY, USA (Jan.), 109–122. https://doi.org/10.1145/174675.176926

Xavier Leroy. 1995. Applicative functors and fully transparent higher-order modules. In *Conference Record of the 22nd Annual ACM Symposium on Principles of Programming Languages (POPL '95)*. Association for Computing Machinery, New York, NY, USA (Jan.), 142–153. https://doi.org/10.1145/199448.199476

Xavier Leroy. 1996. A syntactic theory of type generativity and sharing. *Journal of Functional Programming* 6, 5, 667–698. https://doi.org/10.1017/S0956796800001933

Allen Leung and Lal George. 1999. Static Single Assignment Form for Machine Code. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99)* (Atlanta, GA, USA). Association for Computing Machinery, New York, NY, USA (May), 204–214. https://doi.org/10.1145/301618.301667

Mark Lillibridge. 1997. *Translucent Sums: A Foundation for Higher-Order Module Systems.* Ph.D. Dissertation. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA (May).

Mark Lillibridge and Robert Harper. 1991. ML with callcc is unsound. . Posting to Types mailing list. July 1991. https://www.seas.upenn.edu/~sweirich/types/archive/1991/msg00034.html Archived at Internet Archive: https://web.archive.org/web/20150116061224/https://www.seas.upenn.edu/~sweirich/types/archive/1991/msg00034.html (16 Feb. 2015 06:12:24) Example of ML type unsoundness caused by first-class continuations.

C. H. Lindsey. 1996. *A History of ALGOL 68.* Association for Computing Machinery, New York, NY, USA, 27–96. https://doi.org/10.1145/234286.1057810

Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. 1981. *CLU Reference Manual.* Lecture Notes in Computer Science, Vol. 114. Springer-Verlag, New York, NY, USA. https://doi.org/10.1007/BFb0035014

Barbara Liskov, Alan Snyder, Russell Atkinson, and J. Craig Schaffert. 1977. Abstraction Mechanisms in CLU. *CACM* 20, 8 (Aug.), 564–576. https://doi.org/10.1145/359763.359789

P. Lucas, P. Lauer, and H. Stigleitner. 1968. *Method and notation for the formal definition of programming languages.* Technical Report TR 25.037. IBM Laboratory Vienna, Vienna, Austria (June).

Zhaohui Luo and Randy Pollack. 1992. *LEGO Proof Development System: User's Manual.* Technical Report ECS-LFCS-92-211. LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, UK.

David MacQueen. 1983a. Modified Damas Algorithm for Typechecking with References. March 1983. 6 pages. http://sml-family.org/history/MacQueen-typechecking-refs-1983_03_22.pdf Archived at Internet Archive: https://web.archive.org/web/20200316202127/http://sml-family.org/history/MacQueen-typechecking-refs-1983_03_22.pdf (16 March 2020 20:21:27) Manuscript note dated 1983.03.22.

David MacQueen. 2002. Should ML be Object-Oriented? *Formal Aspects of Computing* 13, 214–232. https://doi.org/10.1007/s001650200010

David MacQueen. 2010. A simple and effective method for assigning blame for type errors. In *Proceedings of the 2010 ACM SIGPLAN Workshop on ML (ML '10)* (Baltimore, MD, USA). Association for Computing Machinery, New York, NY, USA (Sept.), 2.

David MacQueen and Robin Milner. 1985. Record of the Standard ML Meeting, Edinburgh, 6–8 June 1984. *Polymorphism: The ML/LCF/Hope Newsletter* II, 1 (Jan.), 16. http://lucacardelli.name/Papers/Polymorphism%20Vol%20II,%20No%201.pdf Archived at Internet Archive: https://web.archive.org/web/20190225113300/http://lucacardelli.name/Papers/Polymorphism%20Vol%20II,%20No%201.pdf (25 Feb. 2019 11:33:00)

David MacQueen and Mads Tofte. 1994. A semantics for higher order functors. In *Proceedings of the 5th European Symposium on Programming (ESOP '94)* (Edinburg, UK) (*Lecture Notes in Computer Science*), Vol. 788. Springer-Verlag, New York, NY, USA (April), 409–423. https://doi.org/10.1007/3-540-57880-3_27

David B. MacQueen. 1981. Structure and parameterization in a typed functional language. In *The Symposium on Functional Languages and Computer Architecture* (Aspenäs, Sweden). Laboratory for Programming Methodology, Department of Computer Science, University of Göteborg, Göteborg, Sweden (June), 524–538.

David B. MacQueen. 1983b. Modules for Standard ML. *Polymorphism: The ML/LCF/Hope Newsletter* I, 3 (Dec.), 31. http://lucacardelli.name/Papers/Polymorphism%20Vol%20I,%20No%203.pdf Archived at Internet Archive: https://web.archive.org/web/20190307145556/http://lucacardelli.name/Papers/Polymorphism%20Vol%20I,%20No%203.pdf (7 March 2019 14:55:56)

David B. MacQueen. 1984. Modules for Standard ML. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming (LFP '84)* (Austin, TX, USA). Association for Computing Machinery, New York, NY, USA (Aug.), 198–207. https://doi.org/10.1145/800055.802036

David B. MacQueen. 1985a. Dependent Types and Modular Structure. June 1985. http://sml-family.org/history/MacQueen-Marstrand-talk-1985.pdf Archived at Internet Archive: https://web.archive.org/web/20200316201128/http://sml-family.org/history/MacQueen-Marstrand-talk-1985.pdf (16 March 2020 20:11:28) Talk at Workshop on Specification and Derivation of Programs.

David B. MacQueen. 1985b. Modules for Standard ML. *Polymorphism: The ML/LCF/Hope Newsletter* II, 2 (Oct.), 37. http://lucacardelli.name/Papers/Polymorphism%20Vol%20II,%20No%202.pdf Archived at Internet Archive: https://web.

archive.org/web/20190308005136/http://lucacardelli.name/Papers/Polymorphism%20Vol%20II,%20No%202.pdf (8 March 2019 00:51:36) Also in Edinburgh LFCS Tech Report ECS-LFCS-86-2.

David B. MacQueen. 1986. Using dependent types to express modular structure. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages (POPL '86)*. Association for Computing Machinery, New York, NY, USA (Jan.), 277–286. https://doi.org/10.1145/512644.512670

David B. MacQueen. 1994. Reflections on Standard ML. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, Peter E. Lauer (Ed.). Lecture Notes in Computer Science, Vol. 693. Springer-Verlag, New York, NY, USA, 32–46. https://doi.org/10.1007/3-540-56883-2_2

David B. MacQueen. 2015. Notes on Newman's "Stratified systems of logic". May 2015. http://sml-family.org/history/MacQueen-Newman-talk-WG28-2015.pdf Archived at Internet Archive: https://web.archive.org/web/20200317195338/http://sml-family.org/history/MacQueen-Newman-talk-WG28-2015.pdf (17 March 2020 19:53:38) Slides for talk at IFIP Working Group 2.8 meeting, Kefalonia.

B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, and A. van Wijngaarden. 1969. *Report on the Algorithmic Language ALGOL 68*. Springer-Verlag, New York, NY, USA, 80–218. https://doi.org/10.1007/978-3-662-39502-8_1

Luc Maranget. 2008. Compiling Pattern Matching to Good Decision Trees. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML (ML '08)*. Association for Computing Machinery, New York, NY, USA (Sept.), 35–46. https://doi.org/10.1145/1411304.1411311

Per Martin-Löf. 1982. Constructive Mathematics and Computer Programming. In *Logic, Methodology and Philosophy of Science, VI, 1979*. North Holland, Amsterdam, Netherlands, 153–175.

Per Martin-Löf. 1984. *Intuitionistic Type Theory*. Studies in Proof Theory, Vol. 1. Bibliopolis, Napoli, Italy.

David C. J. Matthews. 1988. An Overview of the Poly Programming Language. In *Data Types and Persistence*, Malcolm P. Atkinson, Peter Buneman, and Ronald Morrison (Eds.). Springer-Verlag, New York, NY, USA, 43–50.

David C. J. Matthews. 1989. *Papers on Poly/ML*. Technical Report 161. University of Cambridge Computer Laboratory (Feb.).

Bruce McAdam. 2002. *Repairing Type Errors in Functional Programs*. Ph.D. Dissertation. Laboratory for Foundations of Computer Science, Division of Informatics, University of Edinburgh. LFCS Tech Report ECS-LFCS-02-427.

Bruce J. McAdam. 1998. *On the Unification of Substitutions in Type-Inference*. Technical Report ECS-LFCS-98-384. LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, UK (March).

John McCarthy. 1960. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM* 3, 4 (April), 184–195.

John McCarthy. 1963. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, P. Braffort and D. Hirshberg (Eds.). North Holland, Amsterdam, Netherlands.

John McCarthy. 1966. A formal description of a subset of ALGOL. In *Foraml Language Description Languages for Computer Programming*. North Holland, Amsterdam, Netherlands, 1–12.

Robert Milne. 1974. *The Formal Semantics of Computer Languages and their Implementations*. Ph.D. Dissertation. Oxford University.

Robin Milner. 1972. Implementation and applications of Scott's Logic for Computable Functions. In *Proceedings of the ACM Conference on Proving Assertions about Programs*. Association for Computing Machinery, New York, NY, USA (Jan.), 1–6. https://doi.org/10.1145/942578.807067

Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (Dec.), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

Robin Milner. 1980. *A Calculus of Communicating Systems*. Number 92 in Lecture Notes in Computer Science. Springer-Verlag, New York, NY, USA. https://doi.org/10.1007/3-540-10235-3

Robin Milner. 1983a. Changes to proposal for Standard ML. May 1983. 5 pages. http://sml-family.org/history/SML-changes-4-83.pdf Archived at Internet Archive: https://web.archive.org/web/20170708224109/http://sml-family.org/history/SML-changes-4-83.pdf (8 July 2017 22:41:09)

Robin Milner. 1983b. Discussion with Rod Burstall about precompilation. June 1983. 1 pages. http://sml-family.org/history/Milner-Burstall-precompilation-1983_06_30.pdf Archived at Internet Archive: https://web.archive.org/web/20200317195454/http://sml-family.org/history/Milner-Burstall-precompilation-1983_06_30.pdf (17 March 2020 19:54:54) Manuscript note, dated 1983.6.30.

Robin Milner. 1983c. A Proposal for Standard ML (second draft). June 1983. 50 pages. http://sml-family.org/history/SML-proposal-6-83.pdf Archived at Internet Archive: https://web.archive.org/web/20170705163956/http://sml-family.org/history/SML-proposal-6-83.pdf (5 July 2017 16:39:56)

Robin Milner. 1983d. A Proposal for Standard ML (TENTATIVE). April 1983. 25 pages. http://sml-family.org/history/SML-proposal-4-83.pdf Archived at Internet Archive: https://web.archive.org/web/20170708224143/http://sml-family.org/history/SML-proposal-4-83.pdf (8 Aug. 2017 22:41:43)

Robin Milner. 1984a. ML Meeting Skeleton Timetable. June 1984. 1 pages. http://sml-family.org/history/SML-meeting-84-schedule.pdf Archived at Internet Archive: https://web.archive.org/web/20200317195928/http://sml-family.org/history/SML-meeting-84-schedule.pdf (17 March 2020 19:59:28) proposed timetable for June 84 design meeting.

Robin Milner. 1984b. A proposal for Standard ML. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming (LFP '84)* (Austin, TX, USA). Association for Computing Machinery, New York, NY, USA (Aug.), 184–197. https://doi.org/10.1145/800055.802035

Robin Milner. 1984c. A simplified syntax for instances and module definitions, with a proposal for treating sharing specifications. May 1984. 5 pages. http://sml-family.org/history/Milner-module-notation-1984_05_23.pdf Archived at Internet Archive: https://web.archive.org/web/20200317195607/http://sml-family.org/history/Milner-module-notation-1984_05_23.pdf (17 March 2020 19:56:07) Manuscript note dated 23/5/84, prepared for the June 1984 meeting.

Robin Milner. 1985a. The Dynamic Operational Semantics of Standard ML (3rd Draft). April 1985. 18 pages. http://sml-family.org/history/Milner-dyn-semantics-1985_04.pdf Archived at Internet Archive: https://web.archive.org/web/20200317200028/http://sml-family.org/history/Milner-dyn-semantics-1985_04.pdf (17 March 2020 20:00:28)

Robin Milner. 1985b. Schedule for Standard ML meeting 23–25 May 1985. May 1985. 1 pages. http://sml-family.org/history/SML-meeting-schedule-1985_05.pdf Archived at Internet Archive: https://web.archive.org/web/20200317200130/http://sml-family.org/history/SML-meeting-schedule-1985_05.pdf (17 March 2020 20:01:30) proposed timetable for May 1985 design meeting.

Robin Milner. 1985c. Webs. Sept. 1985. 14 pages. http://sml-family.org/history/Milner-webs-1985_09_01.pdf Archived at Internet Archive: https://web.archive.org/web/20200317001840/http://sml-family.org/history/Milner-webs-1985_09_01.pdf (17 March 2020 00:18:40) "A way of approaching the static semantics of sharing in modules".

Robin Milner. 1986. The Standard ML Core Language (Revised). In *Standard ML*. Number ECS-LFCS-86-2 in LFCS Report Series. LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, UK (March).

Robin Milner. 1987. *Changes to the Standard ML Core language.* Technical Report ECS-LFCS-87-33. LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, UK (Aug.).

Robin Milner and Mads Tofte. 1991. *Commentary on Standard ML.* The MIT Press, Cambridge, MA, USA.

Robin Milner, Mads Tofte, and Robert Harper. 1990. *The Definition of Standard ML.* The MIT Press, Cambridge, MA, USA.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. 1997. *The Definition of Standard ML (Revised).* The MIT Press, Cambridge, MA, USA.

John Mitchell and Ramesh Viswanathan. 1996. Standard ML-NJ Weak Polymorphism and Imperative Constructs. *Information and Computation* 127, 102–116. https://doi.org/10.1006/inco.1996.0054

John C. Mitchell and Robert Harper. 1988. The Essence of ML. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages (POPL '88).* Association for Computing Machinery, New York, NY, USA (Jan.), 28–46. https://doi.org/10.1145/73560.73563

John C. Mitchell and Gordon D. Plotkin. 1988. Abstract types have existential types. *ACM Transactions on Programming Languages and Systems* 10, 3, 470–502. https://doi.org/10.1145/44501.45065

Kevin Mitchell. 1983. ML Progress Report. Sept. 1983. 8 pages. http://sml-family.org/history/KMitchell-Edinburgh-SML-progress-report-1983_09_13.pdf Archived at Internet Archive: https://web.archive.org/web/20200317001939/http://sml-family.org/history/KMitchell-Edinburgh-SML-progress-report-1983_09_13.pdf (17 March 2020 00:19:39) manuscript report to Rod Burstall.

Kevin Mitchell and Robin Milner. 1985. Proposal for I/O in Standard ML. In *ML Workshop '85.* The Standard ML Group, Edinburgh, UK (May), 7. This proposal was originally dated February 1985, but was presented at the ML Workshop in May 1985.

Kevin Mitchell and Alan Mycroft. 1985. The Edinburgh Standard ML Compiler. Jan. 1985. 15 pages. http://sml-family.org/history/KMitchell-Mycroft-Edinburgh%20Standard%20ML%20Compiler-1985_01.pdf Archived at Internet Archive: https://web.archive.org/web/20200317200700/http://sml-family.org/history/KMitchell-Mycroft-Edinburgh%20Standard%20ML%20Compiler-1985_01.pdf (17 March 2020 20:07:00) draft note on the Edinburgh SML compiler.

ML2000 Working Group. 1999. Principles and a preliminary design for ML2000. March 1999. http://flint.cs.yale.edu/flint/publications/ml2000.pdf Archived at Internet Archive: https://web.archive.org/web/20170829123149/http://flint.cs.yale.edu/flint/publications/ml2000.pdf (29 Aug. 2017 12:31:49)

University of Copenhagen 2019. *Try MLKit! The Standard ML Compiler and Toolkit.* University of Copenhagen (July). http://elsman.com/mlkit Archived at Internet Archive: https://web.archive.org/web/20181128041300/http://elsman.com/mlkit/ (28 Nov. 2018 04:13:00)

MLton 2020. *MLton.* MLton (Jan.). http://mlton.org Archived at Internet Archive: https://web.archive.org/web/20200201103035/http://www.mlton.org/ (1 Feb. 2020 10:30:35)

MLton 2014. *MLton History.* MLton (Jan.). http://mlton.org/History Archived at Internet Archive: https://web.archive.org/web/20180212200726/http://mlton.org/History (12 Feb. 2018 20:07:26)

Eugenio Moggi. 1991. A category-theoretic account of program modules. *Mathematical Structures in Computer Science* 1, 1, 103–139. https://doi.org/10.1017/S0960129500000074

Greg Morrisett. 1995. *Compiling with Types.* Ph.D. Dissertation. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA (Dec.). Published as Technical Report CMU-CS-95-226.

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1998. From System F to typed assembly language. In *Conference Record of the 25th Annual ACM Symposium on Principles of Programming Languages (POPL '98)* (San Diego, CA, USA). Association for Computing Machinery, New York, NY, USA, 85–97. https://doi.org/10.1145/319301.319345

J. Gregory Morrisett and Andrew Tolmach. 1993. Procs and Locks: A Portable Multiprocessing Platform for Standard ML of New Jersey. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming.* Association for Computing Machinery, New York, NY, USA (April), 198–207. https://doi.org/10.1145/173284.155353

Moscaow ML 2014. *Moscow ML.* Moscaow ML (Aug.). https://mosml.org Archived at Internet Archive: https://web.archive.org/web/20190514124005/https://mosml.org/ (14 May 2019 12:40:05)

M. H. A. Newman. 1943. Stratified systems of logic. *Proceedings of the Cambridge Philosophical Society* 39, 69–83.

Flemming Nielson (Ed.). 1997. *ML with Concurrency: Design, Analysis, Implementation, and Application.* Springer-Verlag, New York, NY, USA.

INRIA 2019. *A History of OCaml.* INRIA. https://ocaml.org/learn/history.html Archived at Internet Archive: https://web.archive.org/web/20200118031918/https://ocaml.org/learn/history.html (18 Jan. 2020 03:19:18) Accessed 2019:11:06.

Atsushi Ohori. 1992. A Compilation Method for ML-Style Polymorphic Record Calculi. In *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages (POPL '92)* (Albuquerque, NM, USA). Association for Computing Machinery, New York, NY, USA (Jan.), 154–165. https://doi.org/10.1145/143165.143200

Atsushi Ohori. 1995. A Polymorphic Record Calculus and Its Compilation. *ACM Transactions on Programming Languages and Systems* 17, 6 (Nov.), 844–895. https://doi.org/10.1145/218570.218572

Atsushi Ohori, Katsuhiro Ueno, Kazunori Hoshi, Shinji Nozaki, Takashi Sato, Tasuku Makabe, and Yuki Ito. 2014. SML# in Industry: A Practical ERP System Development. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functonal Programming (ICFP '14)* (Gothenburg, Sweden). Association for Computing Machinery, New York, NY, USA (Sept.), 167–173. https://doi.org/10.1145/2692915.2628164

L.C. Paulson. 1996. *ML for the Working Programmer* (2nd ed.). Cambridge University Press, Cambridge, UK.

Zvonimir Pavlinovic, Tim King, and Thomas Wies. 2015. Practical SMT-based type error localization. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functonal Programming (ICFP '15)* (Vancouver, BC, Canada). Association for Computing Machinery, New York, NY, USA (Sept.), 412–423. https://doi.org/10.1145/2784731.2784765

G. Peano. 1889. *Arithmetices Principia, Nova Methodo Exposita.* Fratelli Bocca, Torino, Italy.

Leaf Petersen, Perry Chang, Robert Harper, and Chris Stone. 2000. *Implementing the TILT Internal Language.* Technical Report CMU-CS-00-180. School of Computer Science, Carnegie Mellon University (Dec.).

Leaf Eames Petersen. 2005. *Certifying Compilation for Standard ML in a Type Analysis Framework.* Ph.D. Dissertation. Carnegie Mellon University, Pittsburgh PA (May). Published as technical report CMU-CS-05-135.

Mikael Pettersson. 1992. A term pattern-match compiler inspired by finite automata theory. In *Proceedings of the 4th International Conference on Compiler Construction (CC '92)* (*Lecture Notes in Computer Science*), Vol. 641. Springer-Verlag, New York, NY, USA, 258–270. https://doi.org/10.1007/3-540-55984-1_24

Frank Pfenning and Carsten Schürmann. 1999. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)* (*Lecture Notes in Artificial Intelligence*), H. Ganzinger (Ed.). Springer-Verlag, New York, NY, USA, 202–206. https://doi.org/10.1007/3-540-48660-7_14

Benjamine C. Pierce. 2002. *Types and Programming Languages.* The MIT Press, Cambridge, MA, USA.

Gordon D. Plotkin. 1981. *A structural approach to operational semantics.* Technical Report. Computer Science Department, Aarhus University.

Gordon D. Plotkin. 2004. The Origins of Structural Operational Semantics. *Jounal of Logic and Algebraic Programming* 60-61, 3–15. https://doi.org/10.1016/j.jlap.2004.03.009

School of Computer Science, The University of Birmingham 2019. *Information about Poplog and Pop-11.* School of Computer Science, The University of Birmingham (Dec.). https://www.cs.bham.ac.uk/research/projects/poplog/poplog.info.html Archived at Internet Archive: https://web.archive.org/web/20200201105451/https://www.cs.bham.ac.uk/research/projects/poplog/poplog.info.html (1 Feb. 2020 10:54:51)

R. J. Popplestone. 1968a. The design philosophy of POP-2. In *Machine Intelligence 3*, D. Mitchie (Ed.). Edinburgh University Press, Edinburgh, UK, 393–402.

R. J. Popplestone. 1968b. POP-1: an on-line language. In *Machine Intelligence 2*, E. Dale and D. Mitchie (Eds.). Edinburgh University Press, Edinburgh, UK, 185–94.

François Pottier. 2011. [TYPES] System F omega with (equi-)recursive types. June 2011. 1 pages. http://sml-family.org/history/Pottier-System-F-omega-with-(equi-)recursive-types-2011.pdf Archived at Internet Archive: https://web.archive.org/web/20200317200824/http://sml-family.org/history/Pottier-System-F-omega-with-(equi-)recursive-types-2011.pdf (17 March 2020 20:08:24) message to the Types mailing list.

Francçois Pottier and Didier Rémy. 2005. The Essence of ML Type Inference. In *Advanced Topics in Types and Programming Languages*, Benjamin Pierce (Ed.). The MIT Press, Cambridge, MA, USA, 387–489.

Riccardo Pucella. 2001. *Notes on Programming Standard ML of New Jersey.* Department of Computer Science, Cornell University (Jan.). http://www.cs.cornell.edu/riccardo/prog-smlnj/notes-011001.pdf Archived at Internet Archive: https://web.archive.org/web/20180827154235/http://www.cs.cornell.edu/riccardo/prog-smlnj/notes-011001.pdf (27 Aug. 2018 15:42:35)

F. P. Ramsey. 1926. The foundations of mathematics. *Proceedings of the London Mathematical Society, 2nd Series* 25, 338–384.

Norman Ramsey. 1990. *Concurrent programming in ML.* Technical Report CS-TR-262-90. Department of Computer Science, Princeton University, Princeton, NJ, USA (April).

Norman Ramsey, Kathleen Fisher, and Paul Govereau. 2005. An Expressive Language of Signatures. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP '05)* (Tallinn, Estonia). Association for Computing Machinery, New York, NY, USA (Sept.), 27–40. https://doi.org/10.1145/1090189.1086371

Didier Rémy. 1992. *Extension of ML type system with a sorted equation theory on types.* Technical Report RR-1766. INRIA, Rocquencourt, France (Oct.).

John Reppy, Claudio V. Russo, and Yingqi Xiao. 2009. Parallel Concurrent ML. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)* (Edinburgh, Scotland, UK). Association for Computing Machinery, New York, NY, USA (August–September), 257–268. https://doi.org/10.1145/1596550.1596588

John H. Reppy. 1988. Synchronous operations as first-class values. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '88).* Association for Computing Machinery, New York, NY, USA (June), 250–259. https://doi.org/10.1145/960116.54015

John H. Reppy. 1989. *First-class synchronous operations in Standard ML.* Technical Report TR 89-1068. Department of Computer Science, Cornell University, Ithaca, NY (Dec.).

John H. Reppy. 1991. CML: A higher-order concurrent language. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '91).* Association for Computing Machinery, New York, NY, USA (June), 293–305. https://doi.org/10.1145/113445.113470

John H. Reppy. 1993. *A High-performance Garbage Collector for Standard ML.* Technical Memo BL011261-940329-12TM. AT&T Bell Laboratories (Dec.). https://people.cs.uchicago.edu/~jhr/papers/1993/tm-gc.pdf Archived at Internet Archive: https://web.archive.org/web/20200328124914/https://people.cs.uchicago.edu/~jhr/papers/1993/tm-gc.pdf (28 March 2020 12:49:14)

John H. Reppy. 1996. A safe interface to sockets (Draft). May 1996. 10 pages. http://sml-family.org/history/safe-sockets.pdf Archived at Internet Archive: https://web.archive.org/web/20200313180819/http://sml-family.org/history/safe-sockets.pdf (13 March 2020 18:08:19)

John H. Reppy. 1999. *Concurrent Programming in ML.* Cambridge University Press, Cambridge, UK.

John H. Reppy and Jon G. Riecke. 1996. Simple Objects for Standard ML. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)* (Philadelphia, PA, USA). Association for Computing Machinery, New York, NY, USA (May), 171–180. https://doi.org/10.1145/249069.231412

John C. Reynolds. 1970. GEDANKEN – A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept. *Commun. ACM* 13, 5 (May), 308–319. https://doi.org/10.1145/362349.362364

John C. Reynolds. 1972. Definitional Interpreters for Higher-order Programming Languages. In *Proceedings of the ACM Annual Conference - Volume 2* (Boston, MA, USA). Association for Computing Machinery, New York, NY, USA (Aug.), 717–740. https://doi.org/10.1145/800194.805852

John C. Reynolds. 1974. Towards a theory of type structure. In *Colloquium on Programming* (Paris, France) (*Lecture Notes in Computer Science*), Vol. 19. Springer-Verlag, New York, NY, USA (April), 408–423. https://doi.org/10.1007/3-540-06859-7_148

John A. Robinson. 1965. A machine-oriented logic based on the resolution principle. *Journal of the ACM* 12, 1 (Jan.), 23–41. https://doi.org/10.1145/321250.321253

Andreas Rossberg. 2008. *HaMLet S — To Become or Not Become Successor ML.* Max Planck Institute for Software Systems (April). https://people.mpi-sws.org/~rossberg/hamlet/hamlet-succ-1.3.1S5.pdf Archived at Internet Archive: https://web.archive.org/web/20170330175359/https://people.mpi-sws.org/~rossberg/hamlet/hamlet-succ-1.3.1S5.pdf (30 March 2017 17:53:59)

Andreas Rossberg. 2013a. *Defects in the Revised Definition of Standard ML.* Technical Report. Universität des Saarlandes.

Andreas Rossberg. 2013b. *HaMLet — To Be or Not To Be Standard ML (Version 2.0.0).* Max Planck Institute for Software Systems (Oct.). https://people.mpi-sws.org/~rossberg/hamlet/hamlet-2.0.0.pdf Archived at Internet Archive: https:

//web.archive.org/web/20200328172628/https://people.mpi-sws.org/~rossberg/hamlet/hamlet-2.0.0.pdf (28 March 2020 17:26:28)

Andreas Rossberg. 2015. 1ML – Core and Modules United (F-ing First-Class Modules). In *Proceedings of the 20th ACM SIGPLAN International Conference on Functonal Programming (ICFP '15)*. Association for Computing Machinery, New York, NY, USA (Sept.), 35–47. https://doi.org/10.1145/2858949.2784738

Andreas Rossberg. 2018. 1ML – Core and Modules United. *Journal of Functional Programming* 28, e22 (Dec.), 60. https://doi.org/10.1017/S0956796818000205

Andreas Rossberg, Didier Le Botlan, Guido Tack, Thorsten Brunklaus, and Gert Smolka. 2006. *Alice Through the Looking Glass*. Trends in Functional Programming, Vol. 5. Intellect Books, Bristol, UK, Munich, Germany (Feb.), 79–96.

Andreas Rossberg and Derek Dreyer. 2013. Mixin' Up the ML Module System. *ACM Transactions on Programming Languages and Systems* 35, 1 (April), Article 2, 84 pages. https://doi.org/10.1145/2450136.2450137

Andreas Rossberg, Claudio Russo, and Derek Dreyer. 2015. F-ing modules. *Journal of Functional Programming* 24, 5, 529–607. https://doi.org/10.1017/S0956796814000264

Bertrand Russell. 1967. Letter to Frege. In *From Frege to Gödel*, Jean van Heijenoort (Ed.). Harvard University Press, Cambridge, MA, USA, 124–125.

Bertrand Russell and Alfred North Whitehead. 1910,1912,1913. *Principia Mathematica (3 Volumes)*. Cambridge University Press, Cambridge, UK.

Claudio Russo. 1999a. The Definition of Non-Standard ML (Syntax and Static Semantics). July 1999. http://sml-family.org/history/Russo-defn-nonStandard-ML-2014.pdf Archived at Internet Archive: https://web.archive.org/web/20200317055151/http://sml-family.org/history/Russo-defn-nonStandard-ML-2014.pdf (17 March 2020 05:51:51) Unpublished manuscript.

Claudio Russo. 1999b. Non-dependent Types for Standard ML Modules. In *Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP '99)* (Paris, France) (*Lecture Notes in Computer Science*). Springer-Verlag, New York, NY, USA (September–October), 80–97. https://doi.org/10.1007/10704567_5

Claudio V. Russo. 1998. *Types For Modules*. Ph.D. Dissertation. University of Edinburgh. Edinburgh LFCS Technical Report ECS-LFCS-98-389.

Claudio V. Russo. 2001. Recursive Structures for Standard ML. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01)* (Florence, Italy). Association for Computing Machinery, New York, NY, USA (Sept.), 50–61. https://doi.org/10.1145/507669.507644

Mark Ryan and Martin Sadler. 1992. Valuation Systems and Consequent Relations. In *Background: Mathematical Structures*. Handbook of Logic in Computer Science, Vol. 1. Clarendon Press, Oxford, UK, 1–78.

Bratin Saha and Zhong Shao. 1998. Optimal Type Lifting. In *Types in Compilation Workshop* (Kyoto, Japan) (*Lecture Notes in Computer Science*). Springer-Verlag, New York, NY, USA (March), 156–177. https://doi.org/10.1007/BFb0055517

Nobuo Saito. 1982. ML System on Vax Unix. March 1982. 3 pages. http://sml-family.org/history/Saito-Mlsys-README-1982_03.pdf Archived at Internet Archive: https://web.archive.org/web/20200317201310/http://sml-family.org/history/Saito-Mlsys-README-1982_03.pdf (17 March 2020 20:13:10) README for Saito's Unix port of VAX ML.

Donald Sannella. 1985. A Denotational Semantics for ML Modules. Nov. 1985. http://sml-family.org/history/Sannella-module-sematics-1985_11.pdf Archived at Internet Archive: https://web.archive.org/web/20200317055334/http://sml-family.org/history/Sannella-module-sematics-1985_11.pdf (17 March 2020 05:53:34) Draft of 25 November 1985.

Donald Sannella and Andrzej Tarlecki. 1985. Program Specification and Development in Standard ML. In *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages (POPL '85)*. Association for Computing Machinery, New York, NY, USA (Jan.), 67–77. https://doi.org/10.1145/318593.318614

Thomas Schilling. 2011. Constraint-Free Type Error Slicing. In *Trends in Functional Programming (TFP '11)* (*Lecture Notes in Computer Science*). Springer-Verlag, New York, NY, USA, 1–16. https://doi.org/10.1007/978-3-642-32037-8_1

Moses Schönfinkel. 1924. Über die Bausteine der mathematischen Logik. *Math. Ann.* 92, 305–316. Also available in English as "On the building blocks of mathematical logic" in van Heijenoort, 1967, pp 355–366.

Carsten Schürmann. 2009. The Twelf Proof Assistant. In *Theorem Proving in Higher Order Logics (TPHOLs '09)* (Munich, Germany) (*Lecture Notes in Computer Science*), Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer-Verlag, New York, NY, USA, 79–83. https://doi.org/10.1007/978-3-642-03359-9_7

Dana S. Scott and C. Strachey. 1971. Towards a mathematical semantics for computer languages. In *Proceedings of the Symposium on Computers and Automata* (Brooklyn, NY, USA). Polytechnic Press, New York, NY, USA, 19–46.

Jonathan P. Seldin. 2009. The Logic of Church and Curry. In *Logic from Russell to Church*. Handbook of the History of Logic, Vol. 5. Elsevier, New York, NY, USA, 819–873.

Peter Sestoft. 1996. ML pattern match compilation and partial evaluation. In *Partial Evaluation* (*Lecture Notes in Computer Science*), Vol. 1110. Springer-Verlag, New York, NY, USA, 446–464. https://doi.org/10.1007/3-540-61580-6_22

Zhong Shao. 1997. An Overview of the FLINT/ML Compiler. In *Proceedings of the 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97)* (Amsterdam, The Netherlands). Association for Computing Machinery, New York, NY, USA (June), 10.

Zhong Shao and Andrew W. Appel. 1995. A Type-based Compiler for Standard ML. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '95)* (La Jolla, CA, USA). Association for Computing Machinery, New York, NY, USA (June), 116–129. https://doi.org/10.1145/207110.207123

Zhong Shao and Andrew W. Appel. 2000. Efficient and Safe-for-space Closure Conversion. *ACM Transactions on Programming Languages and Systems* 22, 1 (Jan.), 129–161. https://doi.org/10.1145/345099.345125

Zhong Shao, Christopher League, and Stefan Monnier. 1998. Implementing typed intermediate languages. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*. Association for Computing Machinery, New York, NY, USA (Sept.), 313–323. https://doi.org/10.1145/289423.289460

Joanna Sharrad, Olaf Chitil, and Meng Wang. 2018. Delta Debugging Type Errors with a Blackbox Compiler. In *30th Symposium on Implementation and Application of Functional Languages (IFL '18)* (Lowell, MA, USA). Association for Computing Machinery, New York, NY, USA (Sept.), 13–24. https://doi.org/10.1145/3310232.3310243

Anthony L. Shipman. 2002. Unix System Programming with Standard ML. . 482 pages. http://mlton.org/References.attachments/Shipman02.pdf Archived at Internet Archive: https://web.archive.org/web/20200225113813/http://mlton.org/References.attachments/Shipman02.pdf (25 Feb. 2020 11:38:13)

K. C. Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. 2014. MultiMLton: A multicore-aware runtime for Standard ML. *Journal of Functional Programming* 24, 6 (June), 613–674. https://doi.org/10.1017/S0956796814000161

Marvin Solomon. 1978. Type definitions with parameters (Extended Abstract). In *Conference Record of the 5th Annual ACM Symposium on Principles of Programming Languages (POPL '78)*. Association for Computing Machinery, New York, NY, USA (Jan.), 31–38. https://doi.org/10.1145/512760.512765

T. B. Steel, Jr. (Ed.). 1966. *Formal Language Description Languages for Computer Programming.* North Holland, Amsterdam, Netherlands.

Christopher Stone and Robert Harper. 2000. Deciding Type Equivalence in a Language with Singleton Kinds. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00)* (Boston, MA, USA). Association for Computing Machinery, New York, NY, USA, 214–227. https://doi.org/10.1145/325694.325724

Christopher Allan Stone. 2000. *Singleton Kinds and Singleton Types.* Ph.D. Dissertation. Carnegie Mellon University, Pittsburgh, PA, USA (Aug.). CMU-CS-00-153.

Christopher A. Stone and Robert Harper. 2006. Extensional Equivalence and Singleton Types. *ACM Transactions on Computational Logic* 7, 4 (Oct.), 676–722. https://doi.org/10.1145/1183278.1183281

C. Strachey. 1966. CPL Reference Manual. July 1966. 128 pages. http://www.ancientgeek.org.uk/CPL/CPL_Working_Papers.pdf Archived at Internet Archive: https://web.archive.org/web/20190813125728/http://www.ancientgeek.org.uk/CPL/CPL_Working_Papers.pdf (13 Aug. 2019 12:57:28) Privately circulated. Programming Research Unit, Oxford University.

Christopher Strachey. 1967. Fundamental Concepts in Programming Languages. In *Lecture Notes from the International Summer School in Computer Science* (Copenhagen, Denmark). 38. https://doi.org/10.1023/A:1010000313106 Republished in Higher-Order and Symbolic Computation, 13, 11–49, 2000.

Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. 2004. Improving Type Error Diagnosis. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell (Haskell '04)*. Association for Computing Machinery, New York, NY, USA (Sept.), 80–91. https://doi.org/10.1145/1017472.1017486

David Swasey, Tom Murphy VII, Karl Crary, and Robert Harper. 2006a. A Separate Compilation Extension to Standard ML. In *Proceedings of the 2006 ACM SIGPLAN Workshop on ML (ML '06)* (Portland, OR, USA), Andrew Kennedy and Fran cois Pottier (Eds.). Association for Computing Machinery, New York, NY, USA (Sept.), 32–42. https://doi.org/10.1145/1159876.1159883

David Swasey, Tom Murphy VII, Karl Crary, and Robert Harper. 2006b. *A Separate Compilation Extension to Standard ML (Revised and Expanded).* Technical Report CMU–CS–06–133. Carnegie Mellon University School of Computer Science, Pittsburgh, PA, USA (September).

Walid Taha and Tim Sheard. 2000. MetaML and Multi-Stage Programming with Explicit Annotations. *Theoretical Computer Science* 248, 1–2 (Oct.), 211–242. https://doi.org/10.1016/S0304-3975(00)00053-0

David Tarditi. 1996. *Design and Implementation of Code Optimizations for a Type Directed Compiler for Standard ML.* Ph.D. Dissertation. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA (Dec.). Published as Technical Report CMU-CS-97-108.

D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. 1996. TIL: A Type-directed Optimizing Compiler for ML. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)* (Philadelphia, PA, USA). Association for Computing Machinery, New York, NY, USA (May), 181–192. https://doi.org/10.1145/231379.231414

David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. 2004. TIL: A Type-Directed, Optimizing Compiler for ML. *SIGPLAN Notices* 39, 4 (April), 554–567. https://doi.org/10.1145/989393.989449

Robert D. Tennent. 1977. Language design methods based on semantic principles. *Acta Informatica* 8, 97–112. https://doi.org/10.1007/BF00289243

J. W. Thatcher, E. G. Wagner, and J. B. Wright. 1982. Data type specification: parameterization and the power of specification techniques. *ACM Transactions on Programming Languages and Systems* 4, 4 (Oct.), 711–732. https://doi.org/10.1145/69622.357192

F. Tip and T. B. Dinesh. 2001. A slicing-based approach for locating type errors. *ACM Transactions on Software Engineering and Methodology* 10, 1 (Jan.), 5–55. https://doi.org/10.1145/366378.366379

Mads Tofte. 1987. Polymorphic References Revisited. March 1987. 7 pages. http://sml-family.org/history/Tofte-poly-refs-revisited-1987_03_09.pdf Archived at Internet Archive: https://web.archive.org/web/20200317201835/http://sml-family.org/history/Tofte-poly-refs-revisited-1987_03_09.pdf (17 March 2020 20:18:35)

Mads Tofte. 1988. *Operational Semantics and Polymorphic Type Inference.* Ph.D. Dissertation. Department of Computer Science, Univ. of Edinburgh (May). Available as Technical Report CST-52-88 and ECS-LFCS-88-54.

Mads Tofte. 1989. *Four Lectures on Standard ML.* Technical Report ECS-LFCS-89-73. LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, UK (March).

Mads Tofte. 1990. Type inference for polymorphic references. *Information and Computation* 89, 1 (Nov.), 1–34. 10.1016/0890-5401(90)90018-D

Mads Tofte and Lars Birkedal. 1998. A Region Inference Algorithm. *ACM Transactions on Programming Languages and Systems* 20, 4 (July), 734–767. https://doi.org/10.1145/291891.291894

Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. 2004. A Retrospective on Region-Based Memory Management. *Higher-order and Symbolic Computation* 17, 3, 245–265. https://doi.org/10.1023/B:LISP.0000029446.78563.a4

Mads Tofte and Jean-Pierre Talpin. 1994. Implementation of the Typed Call-by-value λ-calculus Using a Stack of Regions. In *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages (POPL '94)* (Portland, Oregon, USA). Association for Computing Machinery, New York, NY, USA (Jan.), 188–201. https://doi.org/10.1145/174675.177855

Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Information and Computation* 132, 2, 109–176. https://doi.org/10.1006/inco.1996.2613

Andrew Tolmach. 1994. Tag-free Garbage Collection Using Explicit Type Parameters. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming (LFP '94)* (Orlando, FL, USA). Association for Computing Machinery, New York, NY, USA (June), 1–11. https://doi.org/10.1145/182409.182411

Andrew Tolmach and Dino P. Oliva. 1998. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming* 8, 4 (July), 367–412. https://doi.org/10.1017/S0956796898003086

Andrew Peter Tolmach. 1992. *Debugging Standard ML.* Ph.D. Dissertation. Department of Computer Science, Princeton University, Princeton, NJ, USA.

Jeffrey D. Ullman. 1998. *Elements of ML Programming* (ML97 ed.). Prentice-Hall, Upper Saddle River, NJ, USA.

Unknown. 1991. Toward a standard Standard ML. . http://sml-family.org/history/std-charter.pdf Archived at Internet Archive: https://web.archive.org/web/20200313003228/http://sml-family.org/history/std-charter.pdf (13 March 2020 00:32:28) This document was written by someone in the Standard ML of New Jersey group; it was likely David MacQueen.

A. van Wijngaarden et al. 1969. *Report on the Algorithmic Language ALGOL 68.* Technical Report. Mathematisch Centrum, Amsterdam.

Myra VanInwegen. 1996. *The Machine-Assisted Proof of Programming Language Properties.* Ph.D. Dissertation. Univeristy of Pennsylvania. Advisor: Carl Gunter.

M. VanInwegen and E. Gunter. 1993. HOL-ML. In *6th International Workshop on Higher order Logic Theorem Proving and Its Applications* (Vancouver, Canada) (*Lecture Notes in Computer Science*). Springer-Verlag, New York, NY, USA (Aug.), 61–73. https://doi.org/10.1007/3-540-57826-9_125

P. L. Wadler and S. Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages (POPL '89).* Association for Computing Machinery, New York, NY, USA, 60–76. https://doi.org/10.1145/75277.75283

Christopher Wadsworth. 1983. ML, LCF, and HOPE. *Polymorphism: The ML/LCF/Hope Newsletter* I, 1 (Jan.), 5. http://lucacardelli.name/Papers/Polymorphism%20Vol%20I,%20No%201.pdf Archived at Internet Archive: https://web.archive.org/web/20190307022704/http://lucacardelli.name/Papers/Polymorphism%20Vol%20I,%20No%201.pdf (7 March 2019 02:27:04)

D. Waltz. 1975. Understanding line drawings of scenes with shadows. In *Psychology of Computer Vision*, P. H. Winston (Ed.). McGraw-Hill, New York, NY, USA, 19–91.

Mitchell Wand. 1986. Finding the Source of Type Errors. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages (POPL '86).* Association for Computing Machinery, New York, NY, USA (Jan.), 38–43. https://doi.org/10.1145/512644.512648

Jeremy Wazny. 2006. *Type inference and type error diagnosis for Hindley/Milner with extensions.* Ph.D. Dissertation. Computer Science and Software Engineering, The University of Melbourne, Melbourne, Australia (Jan.).

J. B. Wells, Allyn Dimock, Robert Muller, and Franklyn Turbak. 2002. A calculus with polymorphic and polyvariant flow types. *Journal of Functional Programming* 12, 3, 183–227. https://doi.org/10.1017/S0956796801004245

Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A. Acar. 2019. Disentanglement in Nested-Parallel Programs. *Proceedings of the ACM on Programming Languages* 4, POPL (Dec.), Article 47, 32 pages. https://doi.org/10.1145/3371115

Limsoon Wong. 2000. The Functional Guts of the Kleisli Query System. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)* (Montreal, Canada). Association for Computing Machinery, New York, NY, USA (Sept.), 1–10. https://doi.org/10.1145/357766.351241

J. M. Wozencraft and Arthur Evans, Jr. 1970. *Notes on Programming Linguistics.* Technical Report. MIT Department of Electrical Engineering (Feb.).

Andrew Wright. 1995. Simple imperative polymorphism. *Journal of Lisp and Symbolic Computation* 8, 4 (Dec.), 343–355. https://doi.org/10.1007/BF01018828

Andrew K. Wright. 1992. Typing References by Effect Inference. In *Proceedings of the 4th European Symposium on Programming (ESOP '92)* (Rennes, France) (*Lecture Notes in Computer Science*). Springer-Verlag, New York, NY, USA (Feb.), 473–491. https://doi.org/10.1007/3-540-55253-7_28

Andrew K. Wright. 1993. *Polymorphism for Imperative Languages without Imperative Types.* Technical Report TR93-200. Rice University (Feb.).

Andrew K. Wright and Matthias Felleisen. 1991. *A Syntactic Approach to Type Soundness.* Technical Report COMP TR91-160. Rice University (April).

Baijun Wu, John Peter Campora III, and Sheng Chen. 2017. Learning User Friendly Type-Error Messages. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (Oct.), Article 106, 29 pages. https://doi.org/10.1145/3133930

William A. Wulf, Ralph L. London, and Mary Shaw. 1976. An Introduction to the Construction and Verification of Alphard Programs. *IEEE Transactions on Software Engineering* SE-2, 4 (Dec.), 253–265. https://doi.org/10.1109/TSE.1976.233830

Danfeng Zhang and Andrew C. Myers. 2014. Towards general diagnosis of static errors. In *Proceedings of the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* Association for Computing Machinery, New York, NY, USA (Jan.), 569–581. https://doi.org/10.1145/2535838.2535870