ROBERT L. CONSTABLE*

RECENT RESULTS IN TYPE THEORY AND THEIR RELATIONSHIP TO AUTOMATH †

The notion of a *telescope* is basic to Automath's theory structure; telescopes provide the context for theorems. A *dependent record type* is an internal version of a telescope and is used in Nuprl to define theories. This paper shows how A. Kopylov defines these record types in terms of *dependent intersections*, a new type constructor.

Definitional equality and *book equality* are fundamental concepts basic to Automath. In computational type theories these concepts appear in a different form, as computational equalities and as *quotient types.* Questions about these concepts have led to interesting discoveries about types and open problems. This paper presents a new formulation of quotient types by A. Nogin, and an open question about them.

1 INTRODUCTION

Automath, especially AutQE, introduced several of the basic ideas of constructive type theory. De Bruijn's 1968 paper on Automath [12] inspired Dana Scott to write his basic article *Constructive Validity* [32] in 1970. His paper outlined ideas that appear also in Martin-Löf's 1973 paper on type theory, *An Intuitionistic Theory of Types: Predicative Part* [25]. These ideas in turn influenced Nuprl's computational type theory [7], about which I am writing. Several of the basic ideas arose in other forms, e.g. from Intuitionistic higher-order logic [19; 31; 33] and from Girard's basic work on system F [18], which led to the Coq type theory [10; 11].

I will show the influence of two basic ideas from Automath on important concepts in type theory made precise only in the last three years, specifically *dependent intersections* and extensional squash operators. The intersection operator defines *telescopes*, and the squash operator is important in treating some *book equalities.* The concepts discussed here apply to predicative type theories, such as Martin Löf's [26] and Nuprl [7]; and to impredicative ones as well, such as Coq [14]; and to logical frameworks such as Twelf [29].

2 CONTEXTS, TELESCOPES AND DEPENDENT RECORDS

Jeff Zucker's paper "Formalization of Classical Mathematics in Automath" appears in the book *Selected Papers on Automath* [27]. Telescopes are the

*Department of Computer Science, Cornell University. Email rc@cs.cornell.edu

†This work was supported by the DoD Multidisciplinary University Research Initiative (MURI) program administered by the Office of Naval Research under Grant N00014-01-1-0765, and by DARPA under Grant F30602-98-2-0198.

Fairouz Kamareddine (eds.),
Thirty Five Years of Automating Mathematics *1–11.*

topic of Section 10.2 of that paper. Zucker compares them to "generalized Σ" types, which are for us *dependent records.*

A *telescope* is essentially a context of cascading type declarations such as

$$x_1 : A_1, x_2 : A_2(x_1), \ldots, x_n : A_n(x_1, \ldots, x_{n-1}).$$

For example, $G : Type, op : G \times G \to G, id : G$ is a telescope setting the context for theorems about groups in which x_1 is G, A_1 is $Type$, x_2 is op, $A_2(x_1)$ is $G \times G \to G$, and so forth. A sequence of values $a_1, \ldots, a_n$ *fits into the telescope* if $a_1 \in A_1, a_2 \in A_2(a_1)$,

$$a_n \in A_n(a_1, \ldots, a_{n-1}).$$

We are now going to work up to an elegant definition of telescopes starting from familiar ideas. We use the notation $\{x_1 : A_1; \ldots; x_n : A_n\}$ for a *record type* with field selectors x_i and types A_i. This notation is basically from Pascal [22]. For example, $\{n : Nat; x : Real\}$ is a record type whose elements are a natural number named n and a real number named x. If r is the record, then $r.n$ denotes its natural number part and $r.x$ denotes its real part.

A record type can be defined as a Σ-type, or Cartesian product, if we regard the field selectors as external. So $Nat \times Real$ is the type; $r.n$ is an abbreviation for $1of(r)$, the first projection, and $r.x$ for $2of(r)$, the second projection.

2.1 Records as functions

A better definition of the record type incorporates the field selectors internally. Let *Labels* be a discrete type, and consider the subtype $\{n, x\}$ of two distinct labels. Now define a function from $\{n, x\}$ into types by

$$A(n) = Nat,\ A(x) = Real.$$

Then define

$$\{n : Nat; x : Real\} \text{ as } i : \{n, x\} \to A(i),$$

where $i : \{n, x\} \to A(i)$ is the dependent function space, sometimes written as a Π-type, $\Pi i : \{n, x\}.A(i)$.

An even more uniform definition arises if we use all elements of *Label*. We define a record type as the function space $i : Label \to A(i)$, where A maps *Label* into types. Normally only a finite number of labels, say $x_1, \ldots, x_n$, are assigned a non-trivial type. We can capture this idea by using the *top type, Top*, as a default. So we let $A(i) = Top$ for any label $i \in Label$ for which $i \neq x_j \ \ j = 1, \ldots, n$.

The Top type is used in Nuprl and is discussed in *Nuprl's Class Theory and its Applications* [8], as well as in *Dependent Intersections: A New Way of Defining Records in Type Theory* [24]. The basic property of Top is that

it contains all objects, and all objects are equal in it. So every type A is a *subtype* of Top, i.e. $A \sqsubseteq Top$. Generally $A \sqsubseteq B$, provided $a = b$ in A implies that $a = b$ in B.

Another key property is that $A \cap Top = A$, where $A \cap B$ is the type whose elements are those common to both A and B, and whose equality is $a = b$ in $A \cap B$ iff $a = b$ in A and $a = b$ in B. It is called an *intersection* type.

For record types we have $\{x{:}A;\ y{:}B;\ z{:}C\} \sqsubseteq \{x{:}A;\ y{:}B\}$. This follows immediately from the definition of this type as a dependent function space over $Label$. The general subtyping of record types is captured exactly by the *contravariance* in function space subtyping

$$A \sqsubseteq A' \text{ and } B \sqsubseteq B' \text{ implies } (A' \to B) \sqsubseteq (A \to B').$$

This relies on the polymorphic nature of functions in Nuprl; it is not a property of set-theoretic function spaces.

2.2 Dependent record types as dependent intersections

Next we want to define *dependent records.* These are telescopes treated as internal types. A natural notation is

$$\{x_1 : A_1;\ x_2 : A_2(x_1);\ \ldots\ ;\ x_n : A_n(x_1, \ldots, x_{n-1})\}.$$

Kopylov [24] discovered an elegant definition of dependent records in terms of a new type constructor that he discovered: *dependent intersections.* Let A be a type and $B[x]$ be a type for all x of the type A. We define *dependent intersection* $x : A \bigcap B[x]$ as the type containing all elements a from A such that a is also in $B[a]$. This is a type that we might write as $\{x : A | x \in B_x\}$, but $x \in B_x$ is not a well-formed proposition of Nuprl unless it is true.

The dependent intersection is not the same as the intersection of a family of types $\bigcap_{x:A} B[x]$. The latter refers to an intersection of types $B[x]$ for all x in A. The difference between these two type constructors is similar to the difference between dependent products $x : A \times B[x] = \Sigma_{x:A} B[x]$ and the product of a family of types $\Pi_{x:A} B[x] = x : A \to B[x]$.

The ordinary binary intersection is just a special case of a dependent intersection with a constant second argument $A \bigcap B = x : A \bigcap B$.

Let $A = \mathbb{Z}$ and $B[x] = \{y : \mathbb{Z} \mid y > 2x\}$ (i.e. $B[x]$ is a type of all integers y, such that $y > 2x$). Then $x : A \bigcap B[x]$ is a set of all integers, such that $x > 2x$.

Two elements, a and a', are equal in the dependent intersection $x : A \bigcap B[x]$ when they are equal both in A and $B[a]$. The rules for the new type appear in Table 1. The semantics is as in [2].

Now we are going to give a method for constructing record types and their elements. The type is defined by other means in *Nuprl's Class Theory and its Applications* [8], and there are other quite different accounts [5; 30]; here we use Kopylov's dependent intersection.

Table 1. Inference rules for dependent intersection

$$\frac{\Gamma \vdash A\,\mathrm{Type} \qquad \Gamma; x : A \vdash B[x]\,\mathrm{Type}}{\Gamma \vdash (x : A \bigcap B[x])\,\mathrm{Type}} \qquad (TypeFormation)$$

$$\frac{\Gamma \vdash A = A' \qquad \Gamma; x : A \vdash B[x] = B'[x]}{\Gamma \vdash (x : A \bigcap B[x]) = (x : A' \bigcap B'[x])} \qquad (TypeEquality)$$

$$\frac{\Gamma \vdash a \in A \qquad \Gamma \vdash a \in B[a] \qquad \Gamma \vdash x : A \bigcap B[x]\,\mathrm{Type}}{\Gamma \vdash a \in (x : A \bigcap B[x])} \qquad (Introduction)$$

$$\frac{\Gamma \vdash a = a' \in A \qquad \Gamma \vdash a = a' \in B[a] \qquad \Gamma \vdash x : A \bigcap B[x]\,\mathrm{Type}}{\Gamma \vdash a = a' \in (x : A \bigcap B[x])} \qquad (Equality)$$

$$\frac{\Gamma; u : (x : A \bigcap B[x]); \Delta[u]; x : A; y : B[x] \vdash C[x, y]}{\Gamma; u : (x : A \bigcap B[x]); \Delta[u] \vdash C[u; u]} \qquad (Elimination)$$

Records Elements of record types are defined as in [8]. They map labels to the corresponding fields. We can build up these functions component by component starting with an empty record. We write the extension as $r.x := a$; think of assigning a to $r.x$, where x is a label.

$\{\} == \lambda l.l$ (the empty record)

We could pick any function as a definition of an empty record.

$(r.x := a) == (\lambda l.\texttt{if}\ \ l = x\ \ \texttt{then}\ \ a\ \ \texttt{else}\ \ r\ l)$ (field update/extension)

Now we use a common notation for the elements of record types.

$\{x_1 = a_1; \ldots; x_n = a_n\} == \{\}.x_1 := a_1.x_2 := a_2. \ldots .x_n := a_n$

These definitions provide that

$$\{x_1 = a_1; \ldots; x_n = a_n\} = \lambda l.\texttt{if}\ \ l = x_1\ \ \texttt{then}\ \ a_1\ \ \texttt{else}$$
$$\ldots$$
$$\texttt{if}\ \ l = x_n\ \ \texttt{then}\ \ a_n$$

$r.x == r\ x$ (field extraction, $r\,x$ is function application).

Record types We will show how to build finite record types by intersection. First we define a singleton record.

$\{x : A\} == \{x\} \to A$ (single-field record type; == is definitional equality)
where $\{x\} == \{l : Label \mid l = x \in Label\}$ is a singleton set.[1]

$\{R_1; R_2\} == R_1 \bigcap R_2$ (independent concatenation of record types)

Thus $\{n : Nat;\ x : Real\}$ is $\{n : Nat\} \cap \{x : Real\}$. Next we include dependency.

$\{self : R_1; R_2[self]\} == self : R_1 \bigcap R_2[self]$ (dependent concatenation)

Here *self* is a variable bound in R_2. We will usually use the name "self" for this variable and use the shortening $\{R_1; R_2[self]\}$ for this type. Further, we will omit "*self*." in the body of R_2, e.g. we will write just x for $self.x$, when such notation does not lead to misunderstanding.[2]

We assume that this concatenation is a left associative operation, and we will omit inner braces. For example, we will write

$$\{x : A;\ y : B[self];\ z : C[self]\}$$

instead of

$$\{\{\{x : A\}; \{y : B[self]\}\}; \{z : C[self]\}\}.$$

Note that in this expression there are two distinct bound variables *self*. The first one is bound in B, and refers to the record itself as a record of the type $\{x : A\}$. The second *self* is bound in C; it also refers to the same record, but it has type $\{x : A; y : B[self]\}$.

The definition of independent concatenation is just a partial case of dependent concatenation, when R_2 does not depend on *self*. We can also define *right associating records,* a less natural idea, but nevertheless expressible:

$$\{s : x : A; R[s]\} == self : \{x : A\} \bigcap R[self.x]$$

Here x is a variable bound in R that represents a field x. Note that we can α-convert the variable s, but not the label x, e.g., $\{s : x : A; R[s]\} = \{y : x : A; R[y]\}$, but $\{s : x : A; R[s]\} \neq \{y : z : A; R[y]\}$. We will usually use the same name for labels and corresponding bound variables.

[1] $\{x : A \mid P[x]\}$ is a standard type constructor in the Nuprl type theory [7].

[2] Note that Allen's notational devices allow Nuprl to display these terms as written here [3].

This connection is right associative, e.g.,

$$\{s : x : A; t : y : B[s]; z : C[s, t]\}$$

stands for

$$\{s : x : A; \{t : y : B[s]; \{z : C[s, t]\}\}\}.$$

These dependent records can be used to capture theories built in a context. For example, here is how a *monoid* is defined in MetaPRL:

$$\begin{array}{l}\{\ M : Type_i\,;\ op : M \times M \to M; id : M;\\ \quad assocax : \forall x, y, z : M.\ op(x, op(y, z)) = op(op(x, y), z);\\ \quad idax : \forall x : M.\ (op(x, id) = x\ \&\ op(id, x) = x)\ \ \}.\end{array}$$

Here we use the fact that propositions are types — another key Automath contribution.

3 AUTOMATH EQUALITIES AND QUOTIENT TYPES

3.1 Definitional and Computational Equality

The formalization of equality reasoning in Automath is novel, and because equality is so basic to mathematics, the consequences are far-reaching. De Bruijn introduced a calculus of *definitional equality* as basic to reasoning.

Definitional equality is the minimal equality relation generated by α, β-conversion, and δ-conversion (the replacement of the left-hand side of a definition by its right-hand side – *definiendum* by *definiens*). The conversions must respect typing: $(\lambda x : A)(a) = b[a/x]$ only if a is of type A.

There is no internal notation, such as $a \stackrel{\text{D}}{=} a'$, for definitional equality. It is not an explicit relation of any theory expressed in Automath, but substitution of equals for equals in any context is a principle of the Automath language and thus of every theory.

Martin-Löf type theories also use a concept similar to Automath's definitional equality, but these theories relax the typing restriction so that the $(\lambda x.b)(a)$ reduces to $b[a/x]$ regardless of types. In Nuprl this reduction rule generates an equality called *computational equality,* written $a \sim a'$. Doug Howe, in his article *Equality in Lazy Computation Systems* [20], proved a very important theorem: that $a \sim a'$ is a congruence on all closed Nuprl terms; that is, if $a \sim a'$ then $\tau(a) \sim \tau(a')$ for any closed τ.

Howe's result allows $a \sim a'$ to be used as in Automath (indeed until Nuprl 4 there was no internal notation for this equality — called also "squiggle equality").

The difference between $a \stackrel{\text{D}}{=} a'$ of Automath and $a \sim a'$ of Nuprl is that $a \sim a'$ is typeless and includes all computational reduction rules (including integer arithmetic) and reduction rules for the Y combinator and all untyped

lambda terms. This relation greatly facilitates automated reasoning in type theory.

Howe's result is an important discovery about type theory that arose by looking at definitional equality in a computational way. It carries one of de Bruijn's pragmatic insights further in a computational setting.

3.2 Book Equality and Quotient Types

Automath also provides a simple mechanism for defining an equality relation on a type A, denoted $IS(A, x, y)$, meaning x equals y in type A. It is axiomatized as an equivalence relation. In Nuprl, $IS(A, x, y)$ corresponds to $x = y$ *in* A, *the equality relation on a type.* As in Martin-Löf type theory, every Nuprl type comes with an equality relation.

In both Automath and Nuprl, $x \sim y$ implies $x = y$ *in* A for the equality relation defined on A or any other relation defined on A. That is, if $R(x)$ is a proposition,

$$x \sim y \text{ implies } R(x) \Longleftrightarrow R(y).$$

In Automath it is possible to introduce additional equality relations on a type A, say $E(A, x, y)$. The only requirement is that the relation is an equivalence. In Nuprl, changing the equality relation on a type changes the type. So if equivalence relation E is to be the equality on A, then the type is denoted $A//E$, and it is called the *quotient type* of A by E. It is used extensively, in [9] for example, to treat equivalence classes computationally.

The Nuprl rules for $A//E$ are based on these four key semantic notions:

- $A//E = A//F$ iff $E \Longleftrightarrow F$
- $a = b$ in $A//E$ iff $a = b$ in A and $E(a, b)$
- $A \sqsubseteq A//E$
- a function f from A to B belongs to $A//E \to B$ iff $E(x, y)$ implies $f(x) = f(y)$

The rules for $A//E$ do three jobs:

1. They *hide the structure* of E since any equivalent F yields an equal type.
2. They *hide the proof* of E since elements of $A//E$ are simply elements of A, e.g. they are not equivalence classes, and they are not tagged.
3. They *identify* elements of A.

In his article *Quotient Types — A Modular Approach* for TPHOLs 2002 [28], A. Nogin introduces a new formulation of the rules which divides these tasks into separate rules. His organization is clearer than the existing quotient rules, and his new rules are useful in other ways, as his article demonstrates.

One of Nogin's rules introduces a strong notion of collapsing structure. In Nuprl it is possible to *squash* a type, $\downarrow A = \{Top \mid A\}$. If A is empty, then so is $\downarrow A$. If a and b are distinct members of A, they are identified in $\downarrow A$. However,

$$\downarrow A = \downarrow B \text{ iff } A = B.$$

Nogin defines an *extensional squash* operator so that

$$\Downarrow A = \Downarrow B \text{ iff } A \Longleftrightarrow B.$$

The other key element of $A//E$ that Nogin separates into a rule is the *nondeterministic* nature of a function's behavior over $A//E$. Essentially, f can operate on any "element of a virtual equivalence class" of $A//E$. He captures this idea by defining

$$ND == \mathbb{B}//True$$

where $\mathbb{B} = \{tt, ff\}$. In $\mathbb{B}//True$, $tt = ff$. Now let

$$nd_x(a; b) == \textit{if } x \textit{ then } a \textit{ else } b \textit{ fi}.$$

To show that a function f maps $A//E$ into B, we need to know that for $a_1 \in A, a_2 \in A$ and $E(a_1, a_2)$, $f(a_1) = f(a_2)$ in B. This will follow from showing that

$$f(nd_x(a_1; a_2)) \in B \quad \text{for } x \in ND.$$

Nogin's new rule organizes such inference compactly. For example, to establish an implication

$$u : A//E \rightarrow C(u)$$

he shows that

$$x_1 : A, x_2 : A, E(x_1, x_2), y : ND \text{ implies } C(nd_y(x_1; x_2)).$$

Nogin's approach might be useful in Alf and Coq as well as in MetaPRL and Nuprl. It applies to Coq's notion of setoid. It could also be adapted to systems such as PVS with a quotient type added, and it raises an interesting open problem for those type theories, which we discuss next.

3.3 Open Problems

The existing rules for quotient types in MetaPRL or Nuprl do not seem to be sufficient. Let $A \equiv B$ denote *extensional* type equality, i.e. $A \sqsubseteq B$ and $B \sqsubseteq A$; if we then define $\mathbb{Z}_k$ to be $\mathbb{Z}//E_k$ where $E_k(x, y)$ iff there is an m such that $x - y = k \cdot m$, then we can prove that $\mathbb{Z}_2 \cap \mathbb{Z}_3 = \mathbb{Z}_6$ and in general we can prove, using existing rules,

$$\mathbb{Z}_k \cap \mathbb{Z}_n \equiv \mathbb{Z}_{lcm(k,n)}$$

where $lcm(k, n)$ is the least common multiple. But this reasoning relies on properties of $\mathbb{Z}$, and not on general considerations that might establish the following:

* $$A//E \cap A//F \equiv A//E\&F.$$

Is * in fact true for Nuprl? The equalities on the two types are the same, and thus * is true in the PER Semantics of Stuart Allen [2], but the existing rules do not seem to let us prove this. Nogin believes that there is a model of Nuprl's rules in which these types are not equal. What new general rule would let us establish it?

4 CONCLUSION

Modern type theories and logical frameworks owe a great deal to Automath. Its straightforward approach to the natural expression of mathematical discourse was bold for its time, and it exerted a force on computer science, as can be seen in the type theoretic language of major theorem provers such as Alf, Coq, HOL, Isabelle, MetaPRL, Nuprl, PVS, Twelf, and others. Automath was in harmony with the type systems of programming languages such as Algol 68, Pascal, and the major dialects of ML. We also see a significant role for type theory in a rigorous semantics for the basic concepts of object oriented languages such as classes, subtyping, inheritance, and objects themselves [1], and OCaml provides a practical link between objects and types.

I am confident that other basic concepts in Automath, such as its management of contexts and the notion of a "tree of knowledge," will continue to provide insights as the research community in automated mathematics moves toward the integration of the libraries of formal mathematics created in various theorem provers [15].

ACKNOWLEDGEMENTS

I want to thank Stuart Allen, A. Kopylov and A. Nogin for discussions of this article, and Juanita Heyerman for helping prepare it. I appreciate the

constructive remarks and questions from the referees.

BIBLIOGRAPHY

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.

[2] S. F. Allen. A Non-type-theoretic Definition of Martin-Löf's Types. In D. Gries, editor, *Proceedings of the 2^{nd} IEEE Symposium on Logic in Computer Science*, pages 215–224. IEEE Computer Society Press, June 1987.

[3] S. F. Allen. From dy/dx to [ ]p: a matter of notation. In *Proceedings of the Conference on User Interfaces for Theorem Provers*, Eindhoven, The Netherlands, 1998.

[4] H. Barendregt and H. Geuvers. Proof-assistants using dependent type systems. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 1149–1238. Elsevier, 2001.

[5] G. Betarte and A. Tasistro. Extension of Martin Löf's type theory with record types and subtyping. In *Twenty-Five Years of Constructive Type Theory*, chapter 2, pages 21–39. Oxford Science Publications, 1999.

[6] R. Bloo, F. Kamareddine, and R. Nederpelt. The Barendregt cube with definitions and generalised reduction. *Information and Computation*, 126(2):123–143, 1996.

[7] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the NuPRL Development System*. Prentice-Hall, NJ, 1986.

[8] R. L. Constable and J. Hickey. NuPRL's class theory and its applications. In F. L. Bauer and R. Steinbrueggen, editors, *Foundations of Secure Computation*, NATO ASI Series, Series F: Computer & System Sciences, pages 91–116. IOS Press, 2000.

[9] R. L. Constable, P. Jackson, P. Naumov, and J. Uribe. Constructively formalizing automata theory. *Proof, Language and Interaction: Essays in Honour of Robert Milner*, 1998.

[10] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[11] T. Coquand and C. Paulin-Mohring. Inductively defined types, preliminary version. In *COLOG '88, International Conference on Computer Logic*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, Berlin, 1990.

[12] N. G. de Bruijn. The mathematical language Automath: its usage and some of its extensions. In J. P. Seldin and J. R. Hindley, editors, *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 29–61. Springer-Verlag, 1970.

[13] N. G. de Bruijn. A survey of the project Automath. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980.

[14] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. *The Coq Proof Assistant User's Guide*. INRIA, Version 5.8, 1993.

[15] A. Franke and M. Kohlhase. MathWeb, an agent-based communication layer for distributed automated theorem proving. In Ganzinger [16].

[16] H. Ganzinger, editor. *Proceedings of the 16^{th} International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, Berlin, July 7–10 1999. Trento, Italy.

[17] H. Geuvers, R. Pollack, F. Wiedijk, and J. Zwanenburg. The algebraic hierarchy of the FTA project. In *Calculemus 2001 Proceedings*, Siena, Italy, 2001.

[18] J.-Y. Girard. Une extension de l'interpretation de Gödel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. In *2nd Scandinavian Logic Symposium*, pages 63–69. Springer-Verlag, NY, 1971.

[19] W. Howard. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, NY, 1980.

[20] D. J. Howe. Equality in lazy computation systems. In *Proceedings of the 4^{th} IEEE Symposium on Logic in Computer Science*, pages 198–203, Asilomar Conference Center, Pacific Grove, California, June 1989. IEEE, IEEE Computer Society Press.
[21] D. J. Howe. Semantic foundations for embedding HOL in NuPRL. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of *Lecture Notes in Computer Science*, pages 85–101. Springer-Verlag, Berlin, 1996.
[22] K. Jensen and N. Wirth. *PASCAL user manual and report*. Springer-Verlag, New York, 1974.
[23] F. Karareddine and R. P. Nederpelt. A unified approach to type theory through a refined lambda-calculus. *Theoretical Computer Science*, 136(1):183–216, December 1994.
[24] A. Kopylov. Dependent intersection: A new way of defining records in type theory. In *Proceedings of 18^{th} IEEE Symposium on Logic in Computer Science*, 2003. To appear.
[25] P. Martin-Löf. An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73*, pages 73–118. North-Holland, Amsterdam, 1973.
[26] P. Martin-Löf. *Intuitionistic Type Theory*. Number 1 in Studies in Proof Theory, Lecture Notes. Bibliopolis, Napoli, 1984.
[27] R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer. *Selected Papers on Automath*, volume 133 of *Studies in Logic and The Foundations of Mathematics*. Elsevier, Amsterdam, 1994.
[28] A. Nogin. Quotient types: A modular approach. In V. A. Carreño, C. A. Muñoz, and S. Tahar, editors, *Proceedings of the 15^{th} International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002)*, volume 2410 of *Lecture Notes in Computer Science*, pages 263–280. Springer-Verlag, 2002. Available at `http://nogin.org/papers/quotients.html`.
[29] F. Pfenning and C. Schürmann. Twelf — a meta-logical framework for deductive systems. In Ganzinger [16], pages 202–206.
[30] R. Pollack. Dependently typed records for representing mathematical structure. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13^{th} International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 461–478. Springer-Verlag, 2000.
[31] D. Prawitz. *Natural Deduction*. Almquist and Wiksell, Stockholm, 1965.
[32] D. Scott. Constructive validity. In D. L. M. Laudelt, editor, *Symposium on Automatic Demonstration*, volume 5(3) of *Lecture Notes in Mathematics*, pages 237–275. Springer-Verlag, New York, 1970.
[33] W. W. Tait. Intensional interpretation of functionals of finite type. *Journal of Symbolic Logic*, 32(2):189–212, 1967.
[34] F. Wiedijk. A contemporary implementation of Automath. Talk presented at the Workshop on 35 years of Automath, Heriot-Watt University, Edinburgh, Scotland, April 10-13, 2002.
[35] J. Zucker. Formalization of classical mathematics in Automath. In *Colloque International de Logique*, pages 135–145, Paris, 1977. Colloques Internationaux du Centre National de la Recherche Scientifique, CNRS.