

Eliminating Dependent Pattern Matching

Healdene Goguen¹, Conor McBride² and James McKinna³

¹ Google, New York, New York

² School of Computer Science and Information Technology, University of Nottingham

³ School of Computer Science, University of St Andrews

Abstract. This paper gives a reduction-preserving translation from Coquand’s *dependent pattern matching* [4] into a traditional type theory [11] with universes, inductive types and relations and the axiom **K** [22]. This translation serves as a proof of termination for structurally recursive pattern matching programs, provides an implementable compilation technique in the style of functional programming languages, and demonstrates the equivalence with a more easily understood type theory.

Dedicated to Professor Joseph Goguen on the occasion of his 65th birthday.

1 Introduction

Pattern matching is a long-established notation in functional programming [3, 19], combining discrimination on constructors and selection of their arguments safely, compactly and efficiently. Extended to dependent types by Coquand [4], pattern matching becomes still more powerful, managing more complexity as we move from simple inductive datatypes, like **Nat** defined as follows,

$$\mathbf{Nat} : \star = \mathbf{zero} : \mathbf{Nat} \mid \mathbf{suc} (n : \mathbf{Nat}) : \mathbf{Nat}$$

to work with inductive families of datatypes [6] like **Fin**, which is indexed over **Nat** (**Fin** n is an n element enumeration), or **Fin**’s ordering relation, \leq , indexed over indexed data.⁴

$$\begin{aligned} \mathbf{Fin} (n : \mathbf{Nat}) : \star &= \mathbf{fz}_n && : \mathbf{Fin} (\mathbf{suc} \ n) \\ & \mid \mathbf{fs}_n (i : \mathbf{Fin} \ n) && : \mathbf{Fin} (\mathbf{suc} \ n) \\ (i : \mathbf{Fin} \ n) \leq_n (j : \mathbf{Fin} \ n) : \star &= \mathbf{leqz}_{n;j} && : \mathbf{fz}_n \leq_{(\mathbf{suc} \ n)} j \\ & \mid \mathbf{leqs}_{n;i;j} (p : i \leq_n j) : \mathbf{fs}_n \ i \leq_{(\mathbf{suc} \ n)} \mathbf{fs}_n \ j \end{aligned}$$

Pattern matching can make programs and proofs defined over such structures just as simple as for their simply-typed analogues. For example, the proof of transitivity for \leq works just the same for **Fin** as for **Nat**:

$$\begin{aligned} \mathbf{trans} (p : i \leq j; q : j \leq k) : i \leq k \\ \mathbf{trans} \quad \mathbf{leqz}_{n;j} \quad q &\mapsto \mathbf{leqz}_{n;k} \\ \mathbf{trans} (\mathbf{leqs}_{n;i';j'} p') (\mathbf{leqs}_{n;j';k'} q') &\mapsto \mathbf{leqs}_{n;i';k'} (\mathbf{trans} p' q') \end{aligned}$$

⁴ Here we write as subscripts arguments which are usually inferrable; informally, and in practice, we omit them entirely.

There is no such luxury in a traditional type theory [14, 20], where a datatype is equipped only with an *elimination constant* whose type expresses its induction principle and whose operational behaviour is primitive recursion. This paper provides a translation from dependent pattern matching in Coquand’s sense to such a type theory—Luo’s UTT [11], extended with the Altenkirch-Streicher **K** axiom ‘uniqueness of identity proofs’ [22]. Coquand observed that his rules admit **K**; Hofmann and Streicher have shown that **K** does not follow from the usual induction principle for the identity relation [9]. We show that (a variant of) **K** is *sufficient* to bridge the gap: it lets us encode the constructor-based unification which Coquand built directly into his rules.

Our translation here deploys similar techniques to those in [18], but we now ensure both that the translated pattern matching equations hold as *reductions* in the target theory and that the equations can be given a conventional operational semantics [1] directly, preserving termination and confluence. By doing so, we justify pattern matching as a language construct, in the style of ALF [13], without compromising the rôle of the elimination constant in characterising the meaning of data.

An early approximant of our translation was added to the LEGO system [12] and demonstrated at ‘Types 1998’. To date, McBride’s thesis [15] is the only account of it, but there the treatment of the empty program is unsatisfying, the computational behaviour is verified only up to conversion, and the issue of unmatched but trusted terms in pattern matching rules is barely considered.

Our recent work describes the key equipment. The account of elimination in [16] uses a *heterogeneous* equality to express unification constraints over dependently typed data. Hence where Coquand’s pattern matching invokes an *external* notion of unification and of structural recursion, we have built the tools we need *within* type theory [17]. Now, finally, we assemble these components to perform dependent pattern matching by elimination.

Overview The rest of the paper is organised as follows. Section 2 examines pattern matching with dependent types, and develops basic definitions, including that of *specialisation* in patterns, as well as the programs which will eventually be translatable to type theory. The key technical definition here is that of *splitting tree*; novel here is the recording of explicit evidence for impossible case branches. Section 3 describes the target type theory. This is extended by function symbols with defining equations which determine reduction rules, subject to certain conditions. The allowable such function definitions arise from the existence of *valid* splitting trees. Finally, Section 4 shows how such function definitions may be eliminated in favour of closed terms in the type theory with the same reduction behaviour; the valid splitting trees precisely correspond to the terms built from constructor case analysis and structural recursion on inductive families, modulo the heterogeneous equality **Eq**.

2 Dependent Pattern Matching

Let us first take a look at what dependent pattern matching is, and why it is a more subtle notion than its simply typed counterpart. Inductive families gain their precision from the way their constructors have *specialised* return types. For example, the constructors of **Fin** can only make elements of sets whose ‘size’ is non-zero. Consider writing some function $\mathbf{p} (i : \mathbf{Nat}; x : \mathbf{Fin} \ i) : \dots$. Trying to match on x without instantiating i is an error. Rather, one *must* take account of the fact that i is sure to be a **suc**, if \mathbf{p} is to typecheck:

$$\begin{array}{l|l} \dots \not\vdash \mathbf{p} \ i \ \mathbf{fz} & : \mathbf{Nat} & \left| \quad \mathbf{p} \ (\mathbf{suc} \ j) \ \mathbf{fz} & \mapsto \dots \\ \dots \not\vdash \mathbf{p} \ i \ (\mathbf{fs} \ y) & : \mathbf{Nat} & \left| \quad \mathbf{p} \ (\mathbf{suc} \ j) \ (\mathbf{fs} \ y) & \mapsto \dots \end{array}$$

Of course, there need not be any actual check at run time whether these $(\mathbf{suc} \ j)$ patterns match—the type system guarantees that they *must* if the patterns for x do. This is not merely a convenient optimisation, it is a new and necessary phenomenon to consider. For example, we may define the property of ‘being in the image of \mathbf{f} ’ for some fixed $\mathbf{f} : S \rightarrow T$, then equip \mathbf{f} with an ‘inverse’:

$$\begin{array}{ll} \mathbf{lmf} (t:T) : \star = \mathbf{imf} (s:S) : \mathbf{lmf} (\mathbf{f} \ s) & \mathbf{inv} (t:T; p:\mathbf{lmf} \ t) : S \\ & \mathbf{inv} (\mathbf{f} \ s) (\mathbf{imf} \ s) \mapsto s \end{array}$$

The typing rules force us to write $(\mathbf{f} \ s)$ for t , but there is no way in general that we can compute s from t by inverting \mathbf{f} . Of course, we actually get s from the constructor pattern $(\mathbf{imf} \ s)$ for p , together with a *guarantee* that t is $(\mathbf{f} \ s)$.

We have lost the ability to consider patterns for each argument independently. Moreover, we have lost the distinction of patterns as the sub-language of terms consisting only of the *linear constructor forms*, and with this, the interpretation of defining equations as rewriting rules is insufficient. It is not enough just to assign dependent types to conventional programs: specialised patterns change what programs can be.

Let us adapt to these new circumstances, and gain from specialisation, exploiting the information it delivers ‘for free’. For example, in a fully decorated version of the step case of the above definition of the **trans** function,

$$\begin{array}{l} \mathbf{trans}_{(\mathbf{suc} \ n);(\mathbf{fs}_n \ i);(\mathbf{fs}_n \ j);(\mathbf{fs}_n \ k)} (\mathbf{leqs}_{n;i;j} \ p') (\mathbf{leqs}_{n;j;k} \ q') \mapsto \\ \mathbf{leqs}_{n;i;k} (\mathbf{trans}_{n;i;j;k} \ p' \ q') \end{array}$$

it is precisely specialisation that ensures the p' and q' are not arbitrary \leq proofs, but rather appropriate ones, which justify the recursive call to **trans**. Meanwhile, we need not analyse the case

$$\dots \not\vdash \mathbf{trans}_{(\mathbf{suc} \ n);(\mathbf{fs}_n \ i);?;k} (\mathbf{leqs}_{n;i;j} \ p') \mathbf{leqz}_{n;k} : \mathbf{fs} \ i \leq_{(\mathbf{suc} \ n)} k$$

because the two proof patterns demand incompatible specialisations of the middle value upon which they must agree. In general, specialisation is given by the *most general unifier* for the type of the value being analysed and the type of the pattern used to match it. Later, we shall be precise about how this works, but let us first sketch how we address its consequences.

2.1 Patterns with Inaccessible Terms

The key to recovering an operational interpretation for these defining equations is to find the distinction between those parts which *require* constructor matching, and those which merely *report* specialisation. We shall show how to translate the *terms* on the left-hand sides of definitional equations written by the programmer into *patterns* which, following Brady [2], augment the usual linear constructor forms with a representation for the arbitrary terms reported by specialisation and *presupposed* to match.

Definition 1 (Patterns)

$$\begin{array}{lll}
 \text{pat} := x & [x] \Longrightarrow x & \text{AV}(x) \Longrightarrow \{x\} \\
 | \mathbf{c} \text{ pat}^* & [\mathbf{c} \vec{p}] \Longrightarrow \mathbf{c} [\vec{p}] & \text{AV}(\mathbf{c} \vec{p}) \Longrightarrow \text{AV}(\vec{p}) \\
 | \underline{\text{term}} & [\underline{t}] \Longrightarrow t & \text{AV}(\underline{t}) \Longrightarrow \emptyset \\
 \text{lhs} := \mathbf{f} \text{ pat}^* & [\mathbf{f} \vec{p}] \Longrightarrow \mathbf{f} [\vec{p}] & \text{AV}(\mathbf{f} \vec{p}) \Longrightarrow \text{AV}(\vec{p})
 \end{array}$$

We say the terms marked \underline{t} are *inaccessible* to the matcher and may not bind variables. The partial map $\text{AV}(-)$ computes the set of *accessible variables*, where $\text{AV}(\vec{p})$ is the *disjoint* union, $\bigsqcup_i \text{AV}(p_i)$, hence $\text{AV}(-)$ is defined only for linear patterns. The map $[-]$ takes patterns back to terms.

We can now make sense of our **inv** function: its left-hand side becomes

$$\mathbf{inv}(\underline{\mathbf{f} s})(\mathbf{im} s)$$

Matching for these patterns is quite normal, with inaccessible terms behaving like ‘don’t care’ patterns, although our typing rules will always ensure that there is actually no choice! We define **MATCH** to be a partial operation yielding a matching substitution, throwing a **CONFLICT** exception⁵, or failing to make progress only in the case of non-canonical values in a nonempty context.

Definition 2 (Matching) *Matching is given as follows:*

$$\begin{array}{lll}
 \text{MATCH}(x, t) & \Longrightarrow [x \mapsto t] \\
 \text{MATCH}(\mathbf{chalk} \vec{p}, \mathbf{chalk} \vec{t}) & \Longrightarrow \text{MATCHES}(\vec{p}, \vec{t}) \\
 \text{MATCH}(\mathbf{chalk} \vec{p}, \mathbf{cheese} \vec{t}) & \uparrow \text{CONFLICT} \\
 \text{MATCH}(\underline{u}, t) & \Longrightarrow \varepsilon \\
 \\
 \text{MATCHES}(\varepsilon, \varepsilon) & \Longrightarrow \varepsilon \\
 \text{MATCHES}(p; \vec{p}; t; \vec{t}) & \Longrightarrow \text{MATCH}(p, t); \text{MATCHES}(\vec{p}, \vec{t})
 \end{array}$$

So, although definitional equations are not admissible as rewriting rules just as they stand, we can still equip them with an operational model which relies only on constructor discrimination. This much, at least, remains as ever it was.

Before we move on, let us establish a little equipment for working with patterns. In our discussion, we write $p[x]$ to stand for p with an accessible x abstracted. We may thus form the instantiation $p[p']$ if p' is a pattern with variables

⁵ We take **chalk** and **cheese** to stand for an arbitrary pair of distinct constructors.

disjoint from those free in $p[-]$, pasting p' for the accessible occurrence of x and $[p']$ for the inaccessible copies. In particular, $p[\mathbf{c} \vec{y}]$ is a pattern, given fresh \vec{y} . Meanwhile, we shall need to apply specialising substitutions to patterns:

Definition 3 (Pattern Specialisation) *If σ is a substitution from variables Δ to terms over Δ' with $\text{AV}(p) = \Delta \uplus \Delta'$ (making σ idempotent), we define the specialisation σp , lifting σ to patterns recursively as follows:*

$$\begin{array}{lll} \sigma x \implies \underline{\sigma x} & \text{if } x \in \Delta & \sigma(\mathbf{c} \vec{p}) \implies \mathbf{c} \sigma \vec{p} & \sigma \underline{t} \implies \underline{\sigma t} \\ \sigma x \implies x & \text{if } x \in \Delta' & & \end{array}$$

Observe that $\text{AV}(\sigma p) = \Delta'$.

Specialisations, being computed by unification, naturally turn out to be idempotent. Their effect on a pattern variable is thus either to retain its accessibility or to eliminate it entirely, replacing it with an inaccessible term. Crucially, specialisation preserves the availability of a matching semantics despite apparently introducing nonlinearity and non-constructor forms.

2.2 Program Recognition

The problem we address in this paper is to recognize programs as total functions in $\text{UTT}+\mathbf{K}$. Naturally, we cannot hope to decide whether it is possible to construct a functional value exhaustively specified by a set of arbitrary equations. What we can do is fix a recognizable and total fragment of those programs whose case analysis can be expressed as a *splitting tree* of constructor discriminations and whose recursive calls are on *structurally decreasing* arguments.

The idea is to start with a candidate left-hand side whose patterns are just variables and to grow a partition by analysing a succession of pattern variables into constructor cases. This not only gives us an efficient compilation in the style of Augustsson [1], it will also structure our translation, with each node mapping to the invocation of an eliminator. Informally, for **trans**, we build the tree

$$\mathbf{trans} \ p \ q \quad \left\{ \begin{array}{l} \mathbf{trans} \ \text{leqz} \ q \ \mapsto \ \text{leqz} \\ \mathbf{trans} \ (\text{leqs} \ p') \ q \\ \quad q : \text{fs } j \leq k \ \left\{ \begin{array}{l} \mathbf{trans} \ (\text{leqs} \ p') \ \text{leqz} \\ \mathbf{trans} \ (\text{leqs} \ p') \ (\text{leqs} \ q') \ \mapsto \ \text{leqs} \ (\mathbf{trans} \ p' \ q') \end{array} \right. \end{array} \right.$$

The program just gives the leaves of this tree: finding the whole tree guarantees that it partitions the possible input. The recursion reduces the size of one argument (both, in fact, but one is enough), so the function is total.

However, if we take a ‘program’ just to be a set of definitional equations, even this recognition problem is well known to be undecidable [4, 15, 21]. The difficulty for the recognizer is the advantage for the programmer: specialisation can prune the tree! Above, we can see that q must be split to account for $(\text{leqs} \ q')$,

and having split q , we can confirm that no `leqz` case is possible. But consider the signature `empty (i:Fin zero) : X`. We have the splitting tree:

$$\begin{array}{l} \text{empty } i \\ i : \text{Fin zero} \end{array} \left\{ \begin{array}{l} \text{empty fz} \\ \text{empty (fs } i') \end{array} \right.$$

If we record only the leaves of the tree for which we return values, we shall not give the recognizer much to work from! More generally, it is possible to have arbitrarily large splitting trees with no surviving leaves—it is the need to recover these trees from thin air that makes the recognition of equation sets undecidable. Equations are insufficient to define dependently typed functions, so we had better allow our programs to consist of something more. We extend the usual notion of program to allow clauses $\mathbf{f} \vec{t} \uparrow x$ which *refute* a pattern variable, requiring that splitting it leaves no children. For example, we write

$$\begin{array}{l} \text{empty } (i:\text{Fin zero}) \\ \text{empty } i \uparrow i \end{array}$$

We now give the syntax for programs and splitting trees.

Definition 4 (Program, Splitting Tree)

$$\begin{array}{ll} \text{program} := \mathbf{f} (\text{context}) : \text{term} & \text{splitting} := \text{compRule} \\ \quad \text{clause}^+ & \quad | [\text{context}] \text{lhs} \\ \text{clause} := \mathbf{f} \text{term}^* \text{rhs} & \quad \quad \quad x \{ \text{splitting}^+ \\ \text{rhs} := \mapsto \text{term} & \text{compRule} := [\text{context}] \text{lhs rhs} \\ \quad | \uparrow x & \end{array}$$

We say that a splitting tree solves the programming problem $[\Delta] \mathbf{f} \vec{p}$, if these are the context and left-hand side at its root node. Every such programming problem must satisfy $\text{AV}(\vec{p}) = \Delta$, ensuring that every variable is accessible.

To recognize a program with clauses $\{\mathbf{f} \vec{t}_i r_i \mid 0 \leq i \leq n\}$ is to find a valid splitting tree with computation rules $\{[\Delta_i] \mathbf{f} \vec{p}_i r_i \mid 0 \leq i \leq n\}$ such that $[\mathbf{f} \vec{p}_i] = \mathbf{f} \vec{t}_i$ and to check the guardedness of the recursion. We defer the precise notion of ‘valid’ until we have introduced the type system formally, but it will certainly be the case that if an internal node has left-hand side $\mathbf{f} \vec{p}[x]$, then its children (numbering at least one) have left-hand sides $\mathbf{f} \sigma \vec{p}[\mathbf{c} \vec{y}]$ where \mathbf{c} is a constructor and σ is the specialising substitution which unifies the datatype indices of x and $\mathbf{c} \vec{y}$.

We fix unification to be *first-order* with datatype constructors as the rigid symbols [10]—we have systematically shown constructors to be injective and disjoint, and that inductive families do not admit cyclic terms [17]. Accordingly, we have a terminating unification procedure for two vectors of terms which will either *succeed positively* (yielding a specialising substitution), *succeed negatively* (establishing a constructor conflict or cyclic equation), or *fail* because the problem is too hard. Success is guaranteed if the indices are in constructor form.

We can thus determine if a given left-hand side may be split at a given pattern variable—we require all the index unifications to succeed—and generate specialised children for those which succeed positively. We now have:

Lemma 5 (Decidable Coverage) *Given $\mathbf{f}(\vec{x}:\vec{S}) : T$; $\{\mathbf{f} \vec{t}_i r_i \mid 0 \leq i \leq n\}$, it is decidable whether there exists a splitting tree, with root $[\vec{x}:\vec{S}] \mathbf{f} \vec{x} : T$ and computation rules $\{[\Delta_i] \mathbf{f} \vec{p}_i r_i \mid 0 \leq i \leq n\}$ such that $[\mathbf{f} \vec{p}_i] = \mathbf{f} \vec{t}_i$.*

Proof The total number of constructor symbols in the subproblems of a splitting node strictly exceeds those in the node’s problem. We may thus generate all candidate splitting trees whose leaves bear at most the number of constructors in the program clauses and test if any yield the program. \square

Coquand’s specification of a *covering* set of patterns requires the construction of a splitting tree: if we can find a covering for a given set of equations, we may read off one of our programs by turning the childless nodes into refutations. As far as recursion checking is concerned, we may give a criterion a little more generous than Coquand’s original [4].

Definition 6 (Guardedness, Structural Recursion) *We define the binary relation \prec , ‘is guarded by’, inductively on the syntax of terms:*

$$\frac{}{t_i \prec c t_1 \dots t_n} \quad 1 \leq i \leq n \quad \frac{f \prec t}{f s \prec t} \quad \frac{r \prec s \quad s \prec t}{r \prec t}$$

We say that a program $\mathbf{f}(\vec{x}:\vec{S}) : T$; $\{\mathbf{f} \vec{t}_i r_i \mid 0 \leq i \leq n\}$ is structurally recursive if, for some argument position j , we have that every recursive call $\mathbf{f} \vec{s}$ which is a subterm of some r_i satisfies $s_j \prec t_{ij}$.

It is clearly decidable whether a program is structurally recursive in this sense. Unlike Coquand, we do permit one recursive call within the argument of another, although this distinction is merely one of convenience. We could readily extend this criterion to cover lexicographic descent on a number of arguments, but this too is cosmetic. Working in a higher-order setting, we can express the likes of Ackermann’s function, which stand beyond *first-order* primitive recursion. Of course, the interpreter for our own language is beyond it.

3 Type Theory and Pattern Matching

We start from a predicative subsystem of Luo’s UTT [11], with rules of inference given in Figure 1. UTT’s dependent types and inductive types and families are the foundation for dependent pattern matching. Programs with pattern matching are written over types in the base type universe \star_0 , which we call *small* types. Eliminations over types to solve unification are written in \star_1 , and the Logical-Framework-level universe \square is used to define a convenient presentation of equality from the traditional **I**, **J** and **K**. Our construction readily extends to

validity $\boxed{\text{context} \vdash \text{valid}}$ $\frac{}{\mathcal{E} \vdash \text{valid}}$ $\frac{\Gamma \vdash S : \alpha}{\Gamma; x : S \vdash \text{valid}}$ $\alpha \in \{\star_0, \star_1, \square\}$

typing $\boxed{\text{context} \vdash \text{term} : \text{term}}$

$$\frac{\Gamma \vdash \text{valid}}{\Gamma \vdash x : S} \quad x : S \in \Gamma \quad \frac{\Gamma \vdash \text{valid}}{\Gamma \vdash \star_0 : \star_1} \quad \frac{\Gamma \vdash \text{valid}}{\Gamma \vdash \star_1 : \square}$$

$$\frac{\Gamma \vdash S : \alpha \quad \Gamma; x : S \vdash T : \alpha}{\Gamma \vdash \Pi x : S. T : \alpha} \quad \alpha \in \{\star_0, \star_1, \square\}$$

$$\frac{\Gamma; x : S \vdash t : T}{\Gamma \vdash \lambda x : S. t : \Pi x : S. T} \quad \frac{\Gamma \vdash f : \Pi x : S. T \quad \Gamma \vdash s : S}{\Gamma \vdash f s : [x \mapsto s]T}$$

$$\frac{\Gamma \vdash t : S \quad S \preceq T \quad \Gamma \vdash T : \sigma}{\Gamma \vdash t : T}$$

reduction $\boxed{\text{term} \rightsquigarrow \text{term}}$ $\frac{}{(\lambda x : S. t) s \rightsquigarrow [x \mapsto s]t}$ plus contextual closure

conversion $\boxed{\text{term} \cong \text{term}}$ equivalence closure of \rightsquigarrow

cumulativity $\boxed{\text{term} \preceq \text{term}}$

$$\frac{}{\star_0 \preceq \star_1} \quad \frac{}{\star_1 \preceq \square} \quad \frac{S_1 \cong S_2 \quad T_1 \preceq T_2}{\Pi x : S_1. T_1 \preceq \Pi x : S_2. T_2} \quad \frac{S \cong T}{S \preceq T} \quad \frac{R \preceq S \quad S \preceq T}{R \preceq T}$$

Fig. 1. Luo's UTT (functional core)

the additional hierarchy of universes of full UTT. The impredicative universe of propositions in UTT is not relevant to explaining pattern matching through the primitive constructs of type theory, and so we omit it.

We identify terms that are equivalent up to the renaming of bound variables, and we write $[x \mapsto s]t$ for the usual capture-free substitution of s for the free variable x in t .

UTT is presented through the Logical Framework, a meta-language with typed arities for introducing the constants and equalities that define a type theory. While the Logical Framework is essential to the foundational understanding of UTT, it is notationally cumbersome, and we shall hide it as much as possible. We shall not distinguish notationally between framework Π kinds and object-level Π types, nor between the framework and object representations of types. We justify this by observing that \square represents the types in the underlying framework, and that \star_0 and \star_1 are universes with names of specific types within \square . However, informality with respect to universes may lead to size issues if we are not careful, and we shall explicitly mention the cases where it is important to distinguish between the framework and object levels.

There is no proof of the standard metatheoretic properties for the theory UTT plus \mathbf{K} that we take as our target language. Goguen's thesis [8] establishes the metatheory for a sub-calculus of UTT with the Logical Framework, a single

universe and higher-order inductive types but not inductive families or the **K** combinator. Walukiewicz-Chrzaszcz [23] shows that certain higher-order rewrite rules are terminating in the Calculus of Constructions, including inductive families and the **K** combinator, but the rewrite rules do not include higher-order inductive types, and the language is not formulated in the Logical Framework.

However, our primary interest is in justifying dependent pattern matching by translation to a traditional presentation of type theory, and UTT plus **K** serves this role very well. Furthermore, the extensions of additional universes, inductive relations and the **K** combinator to the language used in Goguen’s thesis would complicate the structure of the existing proof of strong normalization but do not seem to represent a likely source of non-termination.

3.1 Telescope Notation

We shall be describing general constructions over dependent datatypes, so we need some notational conveniences. We make considerable use of de Bruijn’s *telescopes* [5]—dependent sequences of types—sharing the syntax of contexts. We also use Greek capitals to stand for them. We may check telescopes (and constrain the universe level α of the types they contain) with the following judgment:

$$\frac{\Gamma \vdash \text{valid}}{\Gamma \vdash \mathcal{E} \ \underline{\text{tele}}(\alpha)} \quad \frac{\Gamma \vdash S : \alpha \quad \Gamma; x : S \vdash \Delta \ \underline{\text{tele}}(\alpha)}{\Gamma \vdash x : S; \Delta \ \underline{\text{tele}}(\alpha)}$$

We use vector notation \vec{t} to stand for sequences of terms, $t_1; \dots; t_n$. We identify the application $f \ t_1; \dots; t_n$ with $f \ t_1 \ \dots \ t_n$. Simultaneous substitutions from a telescope to a sequence are written $[\Theta \mapsto \vec{t}]$, or $[\vec{t}]$ if the domain is clear. Substituting through a telescope textually yields a sequence of typings $t_1 : T_1; \dots; t_n : T_n$ which we may check by iterating the typing judgment. We write $\vec{t} : \Theta$ for the sequence of typings $[\vec{t}]\Theta$, asserting that the t ’s may instantiate Θ . We also let $\Gamma \vdash \sigma \Delta$ assert that σ is a type-correct substitution from Δ to Γ -terms.

We write $\Pi \Delta. T$ to iterate the Π -type over a sequence of arguments, or $\Delta \rightarrow T$ if T does not depend on Δ . The corresponding abstraction is $\lambda \Delta. t$. We also let telescopes stand as the sequence of their variables, so if $f : \Pi \Delta. T$, then $\Delta \vdash f \ \Delta : T$. The empty telescope is \mathcal{E} , the empty sequence, ε .

3.2 Global Declarations and Definitions

A development in our type theory consists of a *global context* Γ containing declarations of datatype families and their constructors, and definitions of function symbols. To ease our translation, we declare global identifiers g with a telescope of arguments and we demand that they are applied to a suitable sequence wherever they are used. Each function $\mathbf{f}(\Delta) : T$ has a nonempty set of computation rules. We extend the typing and reduction rules (now contextualised) accordingly:

$$\frac{g(\Theta) : T \in \Gamma \quad \Gamma; \Delta \vdash \vec{t} : \Theta}{\Gamma; \Delta \vdash g \ \vec{t} : [\vec{t}]T} \quad \mathbf{f} \ \vec{t} \rightsquigarrow_{\Gamma} \theta e \text{ if } [\Delta'] \mathbf{f} \ \vec{p} \mapsto e \in \Gamma \text{ and } \text{MATCHES}(\vec{p}, \vec{t}) \implies \theta$$

We take the following at least to be basic requirements for defined functions.

Definition 7 (Function Criteria) *To extend Γ with $\mathbf{f}(\Delta) : T$ with computation rules $\{[\Delta_i] \mathbf{f} \vec{p}_i r_i \mid 0 \leq i \leq n\}$, we require that:*

- $\Gamma; \Delta \vdash T : \square$,
- the computation rules arise as the leaves of a splitting tree solving $[\Delta] \mathbf{f} \Delta$,
- the corresponding program is structurally recursive,
- if r_i is $\mapsto e_i$, then $\Gamma; \Delta_i \vdash e_i : P_i$.

We shall check basic properties of pattern matching computation shortly, but we first give our notion of data (and hence splitting) a firm basis.

Definition 8 (Inductive Families) *Inductive families with $n \geq 1$ constructors are checked for strict positivity and introduced globally as shown in figure 2. We write $\overline{\mathbf{D}}$ for the telescope $\Xi; z : \mathbf{D} \Xi$.*

$$\frac{\Gamma \vdash \Xi \text{ tele}(\star_0) \quad \{\Gamma | \mathbf{D}(\Xi) : \star_0 \vdash \Delta_i \text{ con}(\vec{u}_i) \mid i \leq n\};}{\Gamma; \mathbf{D}(\Xi) : \star_0; \quad \{c_i(\Delta_i) : \mathbf{D} \vec{u}_i \mid i \leq n\}; \overline{\mathbf{D}} : \star_1} \text{E}_{\mathbf{D}}(\{M_i : \Pi \Delta_i. \text{HYPS}(\Delta_i, \text{BIG}) \rightarrow \star_1 \mid i \leq n\}; \overline{\mathbf{D}}) : \star_1$$

$$\{[\vec{M}; \Delta_i] \text{E}_{\mathbf{D}} \vec{M} \vec{u}_i (c_i \Delta_i) \mapsto M_i \Delta_i \text{RECS}(\Delta_i, \text{E}_{\mathbf{D}} \vec{M}) \mid i \leq n\};$$

$$\text{e}_{\mathbf{D}}(P : \overline{\mathbf{D}} \rightarrow \star_1; \{m_i : \Pi \Delta_i. \text{HYPS}(\Delta_i, \text{LITTLE}(P)) \rightarrow P \vec{u}_i (c \Delta_i) \mid i \leq n\}; \overline{\mathbf{D}}) : P \overline{\mathbf{D}}$$

$$\{[P; \vec{m}; \Delta_i] \text{e}_{\mathbf{D}} P \vec{M} \vec{u}_i (c_i \Delta_i) \mapsto m_i \Delta_i \text{RECS}(\Delta_i, \text{e}_{\mathbf{D}} P \vec{m}) \mid i \leq n\}$$

$$\vdash \text{valid}$$

where $\text{BIG}(_, _) \implies \star_1$ $\text{LITTLE}(P)(\vec{v}, x) \implies P \vec{v} x$

$$\frac{\Gamma; \Theta \vdash \vec{u} : \Xi}{\Gamma | \mathbf{D}(\Xi) : \star_0; \Theta \vdash \varepsilon \text{ con}(\vec{u})} \quad \frac{\Gamma; \Theta \vdash A : \star_0 \quad \Gamma | \mathbf{D}(\Xi) : \star_0; \Theta; a : A \vdash \Delta \text{ con}(\vec{u})}{\Gamma | \mathbf{D}(\Xi) : \star_0; \Theta \vdash a : A; \Delta \text{ con}(\vec{u})}$$

$$\text{HYPS}(\varepsilon, \text{H}) \implies \varepsilon \quad \text{HYPS}(a : A; \Delta, \text{H}) \implies \text{HYPS}(\Delta, \text{H})$$

$$\text{RECS}(\varepsilon, f) \implies \varepsilon \quad \text{RECS}(a : A; \Delta, f) \implies \text{RECS}(\Delta, f)$$

$$\frac{\Gamma; \Theta \vdash \Phi \text{ tele}(\star_0) \quad \Gamma; \Theta; \Phi \vdash \vec{v} : \Xi \quad \Gamma | \mathbf{D}(\Xi) : \star_0; \Theta \vdash \Delta \text{ con}(\vec{u})}{\Gamma | \mathbf{D}(\Xi) : \star_0; \Theta \vdash r : \Pi \Phi. \mathbf{D} \vec{v}; \Delta \text{ con}(\vec{u})}$$

$$\text{HYPS}(r : \Pi \Phi. \mathbf{D} \vec{v}; \Delta, \text{H}) \implies r' : \Pi \Phi. \text{H}(\vec{v}, r \Phi); \text{HYPS}(\Delta, \text{H})$$

$$\text{RECS}(r : \Pi \Phi. \mathbf{D} \vec{v}; \Delta, f) \implies (\lambda \Phi. f \vec{v} (r \Phi)); \text{RECS}(\Delta, f)$$

Fig. 2. Declaring inductive types with constructors

In Luo's presentation [11], each inductive datatype is an inhabitant of \square ; it is then given a *name* in the universe \star_0 . There is a single framework-level eliminator whose kind is much too large for a UTT type. Our presentation is implemented on top: \mathbf{D} really computes Luo's *name* for the type; our UTT eliminators are

readily simulated by the framework-level eliminator. This definition behaves as usual: for `Nat`, we obtain

$$\begin{aligned}
& \mathbf{Nat} : \star_0; \quad \mathbf{zero} : \mathbf{Nat}; \quad \mathbf{suc}(n : \mathbf{Nat}) : \mathbf{Nat}; \\
& \mathbf{ENat}(Z : \star_1; S : \mathbf{Nat} \rightarrow \star_1 \rightarrow \star_1; n : \mathbf{Nat}) : \star_1; \\
& \quad [Z; S] \mathbf{ENat} Z S \mathbf{zero} \quad \mapsto Z \\
& \quad [Z; S; n] \mathbf{ENat} Z S (\mathbf{suc} n) \mapsto S n (\mathbf{ENat} Z S n) \\
& \mathbf{eNat}(P : \mathbf{Nat} \rightarrow \star_1; z : P \mathbf{zero}; s : \Pi n : \mathbf{Nat}. P n \rightarrow P (\mathbf{suc} n); n : \mathbf{Nat}) : P n; \\
& \quad [P; z; s] \mathbf{eNat} P z s \mathbf{zero} \quad \mapsto z \\
& \quad [P; z; s; n] \mathbf{eNat} P z s (\mathbf{suc} n) \mapsto s n (\mathbf{eNat} P z s n)
\end{aligned}$$

Given this, the `Fin` declaration yields the following (we suppress $\mathbf{E}_{\mathbf{Fin}}$):

$$\begin{aligned}
& \mathbf{Fin}(n : \mathbf{Nat}) : \star_0; \quad \mathbf{fz}(n : \mathbf{Nat}) : \mathbf{Fin} (\mathbf{suc} n); \quad \mathbf{fs}(n : \mathbf{Nat}; i : \mathbf{Fin} n) : \mathbf{Fin} (\mathbf{suc} n); \\
& \mathbf{E}_{\mathbf{Fin}} \cdots; \\
& \mathbf{eFin}(P : \Pi n : \mathbf{Nat}. \mathbf{Fin} n \rightarrow \star_1; \\
& \quad z : \Pi n : \mathbf{Nat}. P_{(\mathbf{suc} n)} (\mathbf{fz}_n); \quad s : \Pi n : \mathbf{Nat}; i : \mathbf{Fin} n. P_n i \rightarrow P_{(\mathbf{suc} n)} (\mathbf{fs}_n i); \\
& \quad n : \mathbf{Nat}; i : \mathbf{Fin} n) : P_n i \\
& \quad [P; z; s; n] \mathbf{eFin} P z s (\mathbf{suc} n) (\mathbf{fz}_n) \quad \mapsto z n \\
& \quad [P; z; s; n; i] \mathbf{eFin} P z s (\mathbf{suc} n) (\mathbf{fs}_n i) \mapsto s n i (\mathbf{eFin} P z s n i)
\end{aligned}$$

All of our eliminators will satisfy the function criteria: each has just one split, resulting in specialised, inaccessible patterns for the indices. As the indices may be arbitrary terms, this is not merely convenient but essential. Rewriting with the standard equational laws which accompany the eliminators of inductive families is necessarily confluent.

Meanwhile, empty families have eliminators which refute their input.

$$\frac{\Gamma \vdash \Xi \quad \underline{\mathbf{tele}(\star_0)}}{\Gamma; \mathbf{D}(\Xi) : \star_0; \mathbf{E}_{\mathbf{D}}(\overline{\mathbf{D}}) : \star_1; \quad [\Xi; x] \mathbf{E}_{\mathbf{D}} \Xi x \pitchfork x; \quad \mathbf{e}_{\mathbf{D}}(P : \overline{\mathbf{D}} \rightarrow \star_1; \overline{\mathbf{D}}) : P \overline{\mathbf{D}}; \quad [P; \Xi; x] \mathbf{e}_{\mathbf{D}} P \Xi x \pitchfork x} \vdash \underline{\mathbf{valid}}$$

We have constructed families over elements of sets, but this does not yield ‘polymorphic’ datatypes, parametric in sets themselves. As Luo does, so we may also parametrise a type constructor, its data constructors and eliminators uniformly over a fixed initial telescope of UTT *types*, including \star_0 .

3.3 Valid Splitting Trees and their Properties

In this section, we deliver the promised notion of ‘valid splitting tree’ and show it fit for purpose. This definition is very close to Coquand’s original construction of ‘coverings’ from ‘elementary coverings’ [4]. Our contribution is to separate the empty splits (with explicit refutations) from the nonempty splits (with nonempty subtrees), and to maintain our explicit construction of patterns in linear constructor form with inaccessible terms resulting from specialisation.

Definition 9 (Valid Splitting Tree) A valid splitting tree for $\mathbf{f}(\Delta) : T$ has root problem $[\Delta] \mathbf{f} \Delta$. At each node,

- either we have $\Delta' \vdash e : \llbracket \vec{p} \rrbracket T$ and computation rule

$$[\Delta'] \mathbf{f} \vec{p} \mapsto e$$

- or we have problem $[\Delta^x; x : \mathbf{D} \vec{v}; {}^x \Delta] \mathbf{f} \vec{p}[x]$ and for each constructor $\mathbf{c}(\Delta^c) : \mathbf{D} \vec{u}$, unification succeeds for \vec{u} and \vec{v} , in which case
 - either all succeed negatively, and the node is the computation rule

$$[\Delta^x; x : \mathbf{D} \vec{v}; {}^x \Delta] \mathbf{f} \vec{p}[x] \dot{\vdash} x$$

- or at least one succeeds positively, and the node is a split of form

$$\begin{array}{c} [\Delta^x; x : \mathbf{D} \vec{v}; {}^x \Delta] \mathbf{f} \vec{p}[x] \\ x \{S \end{array}$$

Each positive success yields a pair (Δ', σ) where σ is a most general idempotent unifier for \vec{u} and \vec{v} satisfying $\Delta' \vdash \sigma \Delta^c; \sigma \Delta^x$ and $\text{DOM}(\sigma) \uplus \Delta' = \Delta^c \uplus \Delta^x$, and contributes a subtree to S with root

$$[\Delta'; \sigma[x \mapsto \mathbf{c} \Delta^c] {}^x \Delta] \mathbf{f} \sigma \vec{p}[\mathbf{c} \Delta^c]$$

We shall certainly need to rely on the fact that matching well typed terms yields type-correct substitutions. We must also keep our promise to use inaccessible terms in patterns only where there is no choice.

Definition 10 (Respectful Patterns) For a function $\mathbf{f}(\Delta) : T$, we say that a programming problem $[\Delta'] \mathbf{f} \vec{p}$ has respectful patterns provided

- $\Delta' \vdash [\vec{p}] : \Delta$
- if $\Theta \vdash \vec{a} : \Delta$ and $\text{MATCHES}(\vec{p}, \vec{a}) \implies \theta$, then $\Theta \vdash \theta \Delta'$ and $\theta[\vec{p}] \cong \vec{a}$.

Let us check that valid splitting trees maintain the invariant.

Lemma 11 (Functions have respectful patterns) If $\mathbf{f}(\Delta) : T$ with computation rules $\{[\Delta_i] \mathbf{f} \vec{p}_i r_i \mid 0 \leq i \leq n\}$ satisfies the function criteria, then $[\Delta_i] \mathbf{f} \vec{p}_i$ has respectful patterns.

Proof The root problem $[\Delta] \mathbf{f} \Delta : T$ readily satisfies these properties. We must show that splitting *preserves* them. Given a typical split as above, taking $[\Delta^x; x : \mathbf{D} \vec{v}; {}^x \Delta] \mathbf{f} \vec{p}[x]$ to some $[\Delta'; {}^x \Delta] \mathbf{f} \sigma \vec{p}[\mathbf{c} \Delta^c]$. Let us show the latter is respectful.

We have $\Delta^x; x : \mathbf{D} \vec{v}; {}^x \Delta \vdash [\vec{p}[x]] : \Delta$, hence idempotence of σ yields $\Delta'; x : \mathbf{D} \sigma \vec{v}; \sigma {}^x \Delta \vdash [\sigma \vec{p}[x]] : \Delta$. But $\mathbf{c} \sigma \Delta^c : \mathbf{D} \sigma \vec{u} \cong \mathbf{D} \sigma \vec{v}$, hence $\Delta'; {}^x \Delta \vdash [\sigma \vec{p}[\mathbf{c} \Delta^c]] : \Delta$.

Now suppose $\text{MATCHES}(\sigma \vec{p}[\mathbf{c} \Delta^c], \vec{a}) \implies \phi$ for $\Phi \vdash \vec{a} : \Delta$. For some $\vec{b} : \Delta^c$, we must have $\text{MATCHES}(\vec{p}[x], \vec{a}) \implies \theta; [x \mapsto \mathbf{c} \vec{b}]$. By assumption, the $\vec{p}[x]$ are respectful, so $\Phi \vdash (\theta; [x \mapsto \mathbf{c} \vec{b}])(\Delta^x; x : \mathbf{D} \vec{v}; {}^x \Delta)$, hence $\mathbf{c} \vec{b} : \mathbf{D} \theta \vec{v} = \mathbf{D} [\Delta^c \mapsto \vec{b}] \vec{u}$, and $\theta; [x \mapsto \mathbf{c} \vec{b}][\vec{p}[x]] \cong \vec{a}$. Rearranging, we get $\theta; [\Delta^c \mapsto \vec{b}][\vec{p}[\mathbf{c} \Delta^c]] \cong \vec{a}$.

But $\theta; [\Delta^c \mapsto \vec{b}]y$ unifies \vec{u} and \vec{v} and thus factors as $\theta' \cdot \sigma$ as σ is the most general unifier. By idempotence of σ , θ' and $\theta; [\Delta^c \mapsto \vec{b}]y$ coincide on Δ' . But ϕ coincides with $\theta; [\Delta^c \mapsto \vec{b}]$ on Δ' because they match the same subterms of the \vec{a} , so $\theta; [\Delta^c \mapsto \vec{b}] = \phi \cdot \sigma$, hence $\phi[\sigma\vec{p}[\mathbf{c} \Delta^c]] \cong \vec{a}$. Moreover, we now have $\Phi \vdash (\phi \cdot \sigma)\Delta^c$ and $\Phi \vdash (\phi \cdot \sigma)(\Delta^x; x : \mathbf{D} \vec{v}; {}^x\Delta)$, but idempotence makes Δ' a subcontext of $\sigma(\Delta^c; \Delta^x)$, so $\Phi \vdash \phi(\Delta'; \sigma^x\Delta)$ as required. \square

Lemma 12 (Matching Reduction Preserves Type) *If $\Theta \vdash \mathbf{f} \vec{a} : A$ and $\mathbf{f}(\Delta) : T$ has a computation rule $[\Delta'] \mathbf{f} \vec{p} \mapsto e$ for which $\text{MATCHES}(\vec{p}, \vec{a}) \implies \theta$, then $\Theta \vdash \theta e : A$.*

Proof By inversion of the typing rules, we must have $[\vec{a}]T \preceq A$. By respectfulness, we have $\Theta \vdash \theta\Delta'$ and $\vec{a} \cong \theta[\vec{p}]$. By construction, $\Delta' \vdash e : \llbracket \vec{p} \rrbracket T$, hence $\Theta \vdash \theta e : [\theta[\vec{p}]]T \cong [\vec{a}]T \preceq A$. \square

Lemma 13 (Coverage) *If a function $\mathbf{f}(\Delta) : T$ is given by computation rules $\{[\Delta_i] \mathbf{f} \vec{p}_i r_i : P_i \mid 0 \leq i \leq n\}$, then for any $\Theta \vdash \vec{t} : \Delta$, it is not the case that for each i , $\text{MATCHES}(\vec{p}_i, \vec{t}) \uparrow \text{CONFLICT}$.*

Proof An induction on splitting trees shows that if we have root problem $\mathbf{f} \vec{p}$ and $\text{MATCHES}(\vec{p}, \vec{t}) \implies \theta$ for well typed arguments \vec{t} , matching cannot yield **CONFLICT** at all the leaf patterns. Either the root is the leaf and the result is trivial, or the root has a split at some $x : \mathbf{D} \vec{v}$. In the latter case, we either have θx not in constructor form and matching gets stuck, or $\theta x = \mathbf{c} \vec{b}$ where $\vec{c}(\Delta^c) : \mathbf{D} \vec{v}$, hence unifying \vec{u} and \vec{v} must have succeeded positively yielding some σ for which we have a subtree whose root patterns, $\sigma\vec{p}[\mathbf{c} \Delta^c]$ also match \vec{t} . Inductively, not all of this subtree's leaf patterns yield **CONFLICT**. \square

It may seem a little odd to present coverage as ‘not **CONFLICT** in all cases’, rather than guaranteed progress for *closed* terms. But our result also treats the case of *open* terms, guaranteeing that progress can only be blocked by the presence of non-constructor forms.

Lemma 14 (Canonicity) *For global context Γ , if $\Gamma \vdash t : \mathbf{D} \vec{v}$, with t in normal form, then t is $\mathbf{c} \vec{b}$ for some \vec{b} .*

Proof Select a minimal counterexample. This is necessarily a ‘stuck function’, $\mathbf{f} \vec{a}$. By the above reasoning, we must have some internal node in \mathbf{f} 's splitting tree $[\Delta^x; x : \mathbf{D} \vec{v}; {}^x\Delta] \mathbf{f} \vec{p}[x]$ with $\theta[\vec{p}[x]] \cong \vec{a}$ but $\Gamma \vdash \theta x : \mathbf{D} \theta \vec{v}$ a non-constructor form. But θx is a proper subterm of $\mathbf{f} \vec{a}$, hence a smaller counterexample. \square

Lemma 15 (Confluence) *If every function defined in Γ satisfies the function criteria, then \sim_{Γ} is confluent.*

Proof Function symbols and constructor symbols are disjoint. By construction, splitting trees yield left-hand sides which match disjoint sets of terms. Hence there are no critical pairs. \square

4 Translating Pattern Matching

In this section, we shall give a complete translation from functions satisfying the function criteria and inhabiting small types to terms in a suitable extension of UTT, via the primitive elimination operators for inductive datatypes. We do this by showing how to construct terms corresponding to the splitting trees which give rise to the functions: we show how to represent programming problems as types for which splitting trees deliver inhabitants, and we explain how each step of problem reduction may be realised by a term.

4.1 Heterogeneous equality

We must first collect the necessary equipment. The unification which we take for granted in splitting trees becomes explicit equational reasoning, step by step. We represent problems using McBride’s *heterogeneous* equality [16]:

$$\begin{aligned} & \text{Eq}(s, T : \star_0; s : S; t : T) : \star_1; \quad \text{refl}(R : \star_0; r : R) : \text{Eq}_{RR} r r; \\ & \text{subst}(R : \star_0; s, t : R; q : \text{Eq}_{RR} s t; P : R \rightarrow \star_1; p : P s) : P t; \\ & [R; r; P; p] \text{subst}_{R;T;T}(\text{refl}_R r) P p \mapsto p \end{aligned}$$

Eq is not a standard inductive definition: it permits the expression of heterogeneous equations, but its eliminator subst gives the Leibniz property only for *homogeneous* equations. This is just a convenient repackaging of the traditional homogeneous identity type family \mathbf{I} . The full construction can be found in [15].

It is to enable this construction that we keep equations in \star_1 . We shall be careful to form equations over data sets, but not equality sets. We are unsure whether it is safe to allow equality sets in \star_0 , even though this would not yield an independent copy of \star_0 in \star_0 . At any rate, it is sufficient that we can form equations over data and eliminate data over equations.

We shall write $s \simeq t$ for $\text{Eq}_{ST} s t$ when the types S, T are clear. Furthermore Eq precisely allows us to express equations between *sequences* of data in the same telescope: the constraints which require the specialisation of datatype indices take exactly this form. Note we always have $\overline{D} \text{tele}(\star_0)$, hence if $\vec{s}, \vec{t} : \overline{D}$, we may form the telescope of equations $q_1 : s_1 \simeq t_1; \dots; q_n : s_n \simeq t_n \text{tele}(\star_1)$ which we naturally abbreviate as $\vec{s} \simeq \vec{t}$. Correspondingly, we write $\text{refl } \vec{t} : \vec{t} \simeq \vec{t}$.

4.2 Standard Equipment for Inductive Datatypes

In [17], we show how to equip every datatype with some useful tools, derived from its eliminator, which we shall need in the constructions to come. To summarise,

case_D is just \mathbf{e}_D weakened by dropping the inductive hypotheses.
 $\text{Below}_D(P : \overline{D} \rightarrow \star_1; \overline{D}) : \star_1$ is the ‘course of values’, defined inductively by Giménez [7]; simulated via \mathbf{E}_D , $\text{Below}_D P \Xi z$ computes an iterated tuple type asserting P for every value structurally smaller than z . For Nat we get

$$\begin{aligned} & \text{Below}_{\text{Nat}} P \text{ zero} \mapsto 1 \\ & \text{Below}_{\text{Nat}} P (\text{suc } n) \mapsto \text{Below}_{\text{Nat}} P n \times P n \end{aligned}$$

$\mathbf{below}_D(P : \bar{D} \rightarrow \star_1; p : \Pi \bar{D}. \mathbf{Below}_D P \bar{D} \rightarrow P \bar{D}; \bar{D}) : \mathbf{Below}_D P \bar{D}$ constructs the tuple, given a ‘step’ function, and is simulated via \mathbf{e}_D :

$$\begin{aligned} \mathbf{below}_{\text{Nat}} P p \mathbf{zero} &\mapsto () \\ \mathbf{below}_{\text{Nat}} P p (\mathbf{suc } n) &\mapsto (\lambda b : \mathbf{Below}_{\text{Nat}} P n. (b, p n b)) (\mathbf{below}_{\text{Nat}} P p n) \end{aligned}$$

$\mathbf{rec}_D(P : \bar{D} \rightarrow \star_1; p : \Pi \bar{D}. \mathbf{Below}_D P \bar{D} \rightarrow P \bar{D}; \bar{D}) : P \bar{D}$ is the structural recursion operator for D , given by $\mathbf{rec}_D P p \bar{D} \mapsto p \bar{D} (\mathbf{below}_D P p \bar{D})$

We use \mathbf{case}_D for splitting and \mathbf{rec}_D for recursion. For unification, we need:

$\mathbf{noConfusion}_D$ is the proof that D ’s constructors are injective and disjoint—also a two-level construction, again by example:

$$\begin{aligned} \mathbf{NoConfusion}_{\text{Nat}}(P : \star_1; x, y : \text{Nat}) : \star_1 \\ \mathbf{NoConfusion}_{\text{Nat}} P \mathbf{zero} \mathbf{zero} &\mapsto P \rightarrow P \\ \mathbf{NoConfusion}_{\text{Nat}} P \mathbf{zero} (\mathbf{suc } y) &\mapsto P \\ \mathbf{NoConfusion}_{\text{Nat}} P (\mathbf{suc } x) \mathbf{zero} &\mapsto P \\ \mathbf{NoConfusion}_{\text{Nat}} P (\mathbf{suc } x) (\mathbf{suc } y) &\mapsto (x \simeq y \rightarrow P) \rightarrow P \\ \mathbf{noConfusion}_{\text{Nat}}(P : \star_1; x, y : \text{Nat}; q : x \simeq y) : \mathbf{NoConfusion}_{\text{Nat}} P x y \\ \mathbf{noConfusion}_{\text{Nat}} P \mathbf{zero} \mathbf{zero} (\mathbf{refl } \mathbf{zero}) &\mapsto \lambda p : P. p \\ \mathbf{noConfusion}_{\text{Nat}} P (\mathbf{suc } x) (\mathbf{suc } x) (\mathbf{refl } (\mathbf{suc } n)) &\mapsto \lambda p : x \simeq x \rightarrow P. p (\mathbf{refl } x) \end{aligned}$$

$\mathbf{NoConfusion}_D$ is simulated by two appeals to \mathbf{E}_D ; $\mathbf{noConfusion}_D$ uses \mathbf{subst} once, then \mathbf{case}_D to work down the ‘diagonal’.

$\mathbf{noCycle}_D$ disproves any cyclic equation in D —details may be found in [17].

Lemma 16 (Unification Transitions) *The following (and their symmetric images) are derivable:*

$$\begin{aligned} \mathbf{deletion} \quad m : \Pi \Delta. P \\ &\vdash \lambda \Delta; q. m \Delta \\ &\quad : \Pi \Delta. t \simeq t \rightarrow P \\ \mathbf{solution} \quad m : \Pi \Delta^0. [x \mapsto t] \Pi \Delta^1. P \\ &\vdash \lambda \Delta; q. \mathbf{subst } T t x q (\lambda x. \Pi \Delta^0; \Delta^1. P) m \Delta^0 \Delta^1 \\ &\quad : \Pi \Delta. t \simeq x \rightarrow P \\ &\quad \underline{\text{if}} \Delta \sim \Delta^0; x : T; \Delta^1 \quad \underline{\text{and}} \Delta^0 \vdash t : T \\ \mathbf{injectivity} \quad m : \Pi \Delta. \vec{s} \simeq \vec{t} \rightarrow P \\ &\vdash \lambda \Delta; q. \mathbf{noConfusion } P (\mathbf{chalk } \vec{s}) (\mathbf{cheese } \vec{t}) q (m \Delta) \\ &\quad : \Pi \Delta. \mathbf{c } \vec{s} \simeq \mathbf{c } \vec{t} \rightarrow P \\ \mathbf{conflict} \quad \vdash \lambda \Delta; q. \mathbf{noConfusion } P (\mathbf{chalk } \vec{s}) (\mathbf{cheese } \vec{t}) q \\ &\quad : \Pi \Delta. \mathbf{chalk } \vec{s} \simeq \mathbf{cheese } \vec{t} \rightarrow P \\ \mathbf{cycle} \quad \vdash \lambda \Delta; q. \mathbf{noCycle } P \dots q \dots \\ &\quad : \Pi \Delta. x \simeq \mathbf{c } [\vec{p}[x]] \rightarrow P \end{aligned}$$

Proof By construction. □

4.3 Elimination with Unification

In [16], McBride gives a general technique for deploying operators whose types resemble elimination rules. We shall use this technique repeatedly in our constructions, hence we recapitulate the basic idea here. Extending the previous account, we shall be careful to ensure that the terms we construct not only have the types we expect but also deliver the computational behaviour required to simulate the pattern matching semantics.

Definition 17 (Elimination operator) *For any telescope $\Gamma \vdash \Xi \underline{\text{tele}}(\star_0)$, we define a Ξ -elimination operator to be any*

$$e : \Pi P : \Pi \Xi. \star_1 . (\Pi \Delta_1. P \vec{s}_1) \rightarrow \dots \rightarrow (\Pi \Delta_n. P \vec{s}_n) \rightarrow \Pi \Xi. P \Xi$$

Note that e_D is a \bar{D} -elimination operator; case_D and rec_D are also. We refer to the Ξ as the *targets* of the operator as they indicate what is to be eliminated; we say P is the *motive* as it indicates why; the remaining arguments we call *methods* as they explain how to proceed in each case which may arise. Now let us show how to adapt such an operator to any *specific* sequence of targets.

Definition 18 (Basic analysis) *If e is a Ξ -elimination operator (as above), $\Delta \underline{\text{tele}}(\star_0)$ and $\Delta \vdash T : \star_1$, then for any $\Delta \vdash \vec{t} : \Xi$, the basic e -analysis of $\Pi \Delta. T$ at \vec{t} is the (clearly derivable) judgment*

$$\begin{aligned} m_1 : \Pi \Delta_1; \Delta. \Xi \simeq \vec{t} \rightarrow T; \dots; m_n : \Pi \Delta_n; \Delta. \Xi \simeq \vec{t} \rightarrow T \\ \vdash \lambda \Delta. e (\lambda \Xi. \Pi \Delta. \Xi \simeq \vec{t} \rightarrow T) m_1 \dots m_n \vec{t} \Delta (\text{refl } \vec{t}) : \Pi \Delta. T \end{aligned}$$

Notice that when e is case_D and the targets are some $\vec{v}; x$ where $x : D \vec{v} \in \Delta$, then for each constructor $c : \Delta^c : D \vec{u}$, we get a method

$$m_c : \Pi \Delta^c; \Delta. \vec{u} \simeq \vec{v} \rightarrow c \Delta^c \simeq x \rightarrow T$$

Observe that the equations on the indices are exactly those we must unify to allow the instantiation of x with $c \Delta^c$. Moreover, if we have such an instance for x , i.e. if θ unifies \vec{u} and \vec{v} , and takes $x \mapsto c \theta \Delta^c$, then the analysis actually reduces to the relevant method:

$$\begin{aligned} \text{case}_D (\lambda \bar{D}. \Pi \Delta. \Xi \simeq \vec{t} \rightarrow T) \vec{m} \theta \vec{v} (c \theta \Delta^c) \theta \Delta (\text{refl } \theta \vec{v}) (\text{refl } (c \theta \Delta^c)) \\ \rightsquigarrow m_c \theta \Delta^c \theta \Delta (\text{refl } \theta \vec{v}) (\text{refl } (c \theta \Delta^c)) \end{aligned}$$

We may now simplify the equations in the method types.

Definition 19 (Specialisation by Unification) *Given any type of the form $\Pi \Delta. \vec{u} \simeq \vec{v} \rightarrow T : \star_1$, we may seek to construct an inhabitant—a specialiser—by exhaustively iterating the unification transitions from lemma 16 as applicable. This terminates by the usual argument [10], with three possible outcomes:*

negative success a specialiser is found, either by *conflict* or *cycle*;

positive success a specialiser is found, given some $m : \Pi \Delta'. \sigma T$ for σ a most general idempotent unifier of \vec{u} and \vec{v} , or
failure at some stage, an equation is reached for which no transition applies.

Lemma 20 (Specialiser Reduction) *If specialisation by unification delivers*

$$m : \Pi \Delta'. \sigma T \vdash s : \Pi \Delta. \vec{u} \simeq \vec{v} \rightarrow T$$

then for any $\Theta \vdash \theta \Delta$ unifying \vec{u} and \vec{v} we have $s \theta \Delta (\text{refl } \theta \vec{u}) \rightsquigarrow^ m \theta \Delta'$.*

Proof By induction on the transition sequence. The **deletion**, **solution** and **injectivity** steps each preserve this property by construction. \square

We can now give a construction which captures our notion of *splitting*.

Lemma 21 (Splitting Construction) *Suppose $\Delta \vdash T : \star_1$, with $\Delta \text{tele}(\star_0)$, $\Delta^x; x : \mathbf{D} \vec{v}; {}^x \Delta \text{tele}(\star_0)$ and $\Delta^x; x : \mathbf{D} \vec{v}; {}^x \Delta \vdash [\bar{p}[x]] : \Delta$. Suppose further that for each $\mathbf{c}(\Delta^c) : \mathbf{D} \vec{u}$, unifying \vec{u} with \vec{v} succeeds. Then we may construct an inhabitant $f : \Pi \Delta^x; x : \mathbf{D} \vec{v}; {}^x \Delta. \llbracket \bar{p}[x] \rrbracket T$ over a context comprising, for each \mathbf{c} with positive success,*

$$m_{\mathbf{c}} : \Pi \Delta'; \sigma[x \mapsto \mathbf{c} \Delta^c] {}^x \Delta. \llbracket \sigma \bar{p}[\mathbf{c} \Delta^c] \rrbracket T$$

for some most general idempotent unifier $\Delta' \vdash \sigma(\Delta^c; \Delta^x)$. In each such case,

$$f \sigma \Delta^x (\mathbf{c} \sigma \Delta^c) {}^x \Delta \rightsquigarrow^* m_{\mathbf{c}} \Delta' {}^x \Delta$$

Proof The construction is by basic **case_D**-analysis of $\Pi \Delta^x; x : \mathbf{D} \vec{v}; {}^x \Delta. \llbracket \bar{p}[x] \rrbracket T$ at $\vec{v}; x$, then specialisation by unification for each method. The required reduction behaviour follows from lemma 20. \square

4.4 Translating Structural Recursion

We are very nearly ready to translate whole functions. For the sake of clarity, we introduce one last piece of equipment:

Definition 22 (Computation Types) *When implementing a function $\mathbf{f}(\Delta) : T$, we introduce the family of **f**-computation types as follows:*

$$\begin{aligned} \text{Comp-}\mathbf{f}(\Delta) : \star_0; \quad \text{return-}\mathbf{f}(\Delta; t : T) : \text{Comp-}\mathbf{f} \Delta \\ \text{call-}\mathbf{f}(\overline{\text{Comp-}\mathbf{f}}) : T \\ \text{call-}\mathbf{f} \underline{\Delta} (\text{return-}\mathbf{f} \Delta t) \mapsto t \end{aligned}$$

*where **call-f** is clearly definable from $\mathbf{e}_{\text{Comp-}\mathbf{f}}$.*

Comp-f book-keeps the connection between **f**'s high-level program and the low-level term which delivers its semantics. We translate each **f**-application to the corresponding **call-f** of an **f**-computation; the latter will compute to a **return-f** value exactly in correspondence with the pattern matching reduction. The translation takes the following form:

Definition 23 (Application Translation) If $\mathbf{f}(\Delta) : T$ is globally defined, but $\Delta \vdash f : \mathbf{Comp}\text{-}\mathbf{f} \Delta$ for some f not containing \mathbf{f} , the translation $\{-\}_{\mathbf{f}}^f$ takes

$$\{\mathbf{f} t\}_{\mathbf{f}}^f \implies \mathbf{call}\text{-}\mathbf{f} \{\vec{t}\}_{\mathbf{f}}^f ((\{\vec{t}\}_{\mathbf{f}}^f) f)$$

and proceeds structurally otherwise. Recalling that we require global functions to be applied with at least their declared arity, this translation removes \mathbf{f} entirely.

Theorem 24 If $\mathbf{f}(\Delta) : T$ has a small type and computation rules $[\Delta_i] \mathbf{f} \vec{p}_i r_i$ satisfying the function criteria, then there exists an f such that

$$\Delta \vdash f : \mathbf{Comp}\text{-}\mathbf{f} \Delta \quad \text{and} \quad s \rightsquigarrow_{\Gamma, \mathbf{f}} t \quad \text{implies} \quad \{s\}_{\mathbf{f}}^f \rightsquigarrow_{\Gamma}^+ \{t\}_{\mathbf{f}}^f$$

Proof It suffices to ensure that the pattern matching reduction schemes are faithfully translated. For each i such that r_i returns a value $\mapsto e_i$, we shall have

$$\{\mathbf{f} [\vec{p}_i]\}_{\mathbf{f}}^f = \mathbf{call}\text{-}\mathbf{f} [\vec{p}_i] [\vec{p}_i] f \rightsquigarrow_{\Gamma}^* \mathbf{call}\text{-}\mathbf{f} [\vec{p}_i] (\mathbf{return}\text{-}\mathbf{f} [\vec{p}_i] \{e_i\}_{\mathbf{f}}^f) \rightsquigarrow_{\Gamma} \{e_i\}_{\mathbf{f}}^f$$

Without loss of generality, let \mathbf{f} be structurally recursive on some $x : \mathbf{D} \vec{v}$, j th in Δ . The basic $\mathbf{rec}_{\mathbf{D}}$ -analysis of $\Pi \Delta$. $\mathbf{Comp}\text{-}\mathbf{f} \Delta$ at $\vec{v}; x$ requires a term of type

$$\Pi \bar{\mathbf{D}}. \mathbf{Below}_{\mathbf{D}} P \bar{\mathbf{D}} \rightarrow \Pi \Delta. \bar{\mathbf{D}} \simeq \vec{v}; x \rightarrow \mathbf{Comp}\text{-}\mathbf{f} \Delta$$

where $P = \lambda \bar{\mathbf{D}}. \Pi \Delta. \bar{\mathbf{D}} \simeq \vec{v}; x \rightarrow \mathbf{Comp}\text{-}\mathbf{f} \Delta$. Specialisation substitutes $\vec{v}; x$ for $\bar{\mathbf{D}}$, yielding a specialiser $[m]s$ of the required type, with

$$\begin{aligned} m : \Pi \Delta. \mathbf{Below}_{\mathbf{D}} P \vec{v} x \rightarrow \mathbf{Comp}\text{-}\mathbf{f} \Delta; \Delta \vdash \mathbf{rec}_{\mathbf{D}} P [m]s \vec{v} x \Delta (\mathbf{refl} \vec{v}; x) \\ \rightsquigarrow_{\Gamma}^* m \Delta (\mathbf{below}_{\mathbf{D}} P [m]s \vec{v} x) : \mathbf{Comp}\text{-}\mathbf{f} \Delta \end{aligned}$$

by definition of $\mathbf{rec}_{\mathbf{D}}$ and specialisation reduction. We shall take the latter to be our f , once we have suitably instantiated m . To do so, we follow \mathbf{f} 's splitting tree: lemma 21 justifies the splitting construction at each internal node and at each $\cap y$ leaf. Each programming problem $[\Delta'] \mathbf{f} \vec{p}$ in the tree corresponds to the task of instantiating some $m' : \Pi \Delta'. \mathbf{Below}_{\mathbf{D}} P ([\vec{p}'](\vec{v}; x)) \rightarrow \mathbf{Comp}\text{-}\mathbf{f} [\vec{p}]$ where, again by lemma 21, $m [\vec{p}] \rightsquigarrow_{\Gamma}^* m' \Delta'$.

The splitting finished, it remains to instantiate the m_i corresponding to each $[\Delta_i] \mathbf{f} \vec{p}_i \mapsto e_i$. Now, $[\Delta \mapsto [\vec{p}_i]]$ takes $x : \mathbf{D} \vec{v}$ to some $[p_{ij}] : \mathbf{D} \vec{u}$, so we may take

$$m_i \mapsto \lambda \Delta_i; H : \mathbf{Below}_{\mathbf{D}} P \vec{u} [p_{ij}]. \mathbf{return}\text{-}\mathbf{f} [\vec{p}_i] e_i^\dagger$$

where e_i^\dagger is constructed by replacing each call $\mathbf{f} \vec{r}$ in e_i by an appropriate appeal to H . As \mathbf{f} is well typed and structurally recursive, so $[\Delta \mapsto \vec{r}]$ maps $x : \mathbf{D} \vec{v}$ to $r_j : \mathbf{D} \vec{w}$ where $r_j \prec [p_{ij}]$. By construction, $\mathbf{Below}_{\mathbf{D}} P \vec{u} [p_{ij}]$ reduces to a tuple of the computations for subobjects of $[p_{ij}]$. Hence we have a projection g such that $g H : \Pi \Delta. \vec{w}; r_j \simeq \vec{v}; x \rightarrow \mathbf{Comp}\text{-}\mathbf{f} \Delta$ and hence we take $\mathbf{call}\text{-}\mathbf{f} \vec{r}(g H \vec{r}(\mathbf{refl} \vec{w}; r_j))$ to replace $\mathbf{f} \vec{r}$, where by construction of $\mathbf{below}_{\mathbf{D}}$,

$$\begin{aligned} \mathbf{call}\text{-}\mathbf{f} \vec{r}(g (\mathbf{below}_{\mathbf{D}} P [m]s \vec{u} [p_{ij}]) \vec{r}(\mathbf{refl} \vec{w}; r_j)) \\ \rightsquigarrow_{\Gamma}^* \mathbf{call}\text{-}\mathbf{f} \vec{r}([m]s \vec{w} r_j \vec{r}(\mathbf{refl} \vec{w}; r_j)) \\ \rightsquigarrow_{\Gamma}^* \mathbf{call}\text{-}\mathbf{f} \vec{r}(m \vec{r}(\mathbf{below}_{\mathbf{D}} P [m]s \vec{w} r_j)) \\ = \{\mathbf{f} \vec{r}\}_{\mathbf{f}}^f \end{aligned}$$

So, finally, we arrive at

$$\begin{aligned}
\{\mathbf{f} [\vec{p}_i]\}_{\mathbf{f}}^f &= \mathbf{call-f} [\vec{p}_i] (m [\vec{p}_i] (\mathbf{below}_D P [m]s \vec{u} [p_{ij}])) \\
&\rightsquigarrow_{\Gamma}^* \mathbf{call-f} [\vec{p}_i] (m_i \Delta_i (\mathbf{below}_D P [m]s \vec{u} [p_{ij}])) \\
&\rightsquigarrow_{\Gamma}^* \mathbf{call-f} [\vec{p}_i] (\mathbf{return-f} [\vec{p}_i] [H \mapsto \mathbf{below}_D P [m]s \vec{u} [p_{ij}]] e_i^\dagger) \\
&\rightsquigarrow_{\Gamma}^* \mathbf{call-f} [\vec{p}_i] (\mathbf{return-f} [\vec{p}_i] \{e_i\}_{\mathbf{f}}^f) \\
&= \{e_i\}_{\mathbf{f}}^f
\end{aligned}$$

as required. \square

5 Conclusions

We have shown that dependent pattern matching can be translated into a powerful though notationally minimal target language. This constitutes the first proof that dependent pattern matching is equivalent to type theory with inductive types extended with the **K** axiom, at the same time reducing the problem of the termination of pattern matching as a first-class syntax for structurally recursive programs and proofs to the problem of termination of UTT plus **K**.

Two of the authors have extended the raw notion of pattern matching that we study here with additional language constructs for more concise, expressive programming with dependent types [18]. One of the insights from that work is that the technology for explaining pattern matching and other programming language constructs is as important as the language constructs themselves, since the technology can be used to motivate and explain increasingly powerful language constructs.

References

1. Lennart Augustsson. Compiling Pattern Matching. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*, pages 368–381. Springer-Verlag, 1985.
2. Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs, Torino, 2003*, volume 3085 of *LNCS*, pages 115–129. Springer-Verlag, 2004.
3. Rod Burstall. Proving properties of programs by structural induction. *Computer Journal*, 12(1):41–48, 1969.
4. Thierry Coquand. Pattern Matching with Dependent Types. In Bengt Nordström, Kent Petersson, and Gordon Plotkin, editors, *Electronic Proceedings of the Third Annual BRA Workshop on Logical Frameworks (Båstad, Sweden)*, 1992.
5. Nicolas G. de Bruijn. Telescopic Mappings in Typed Lambda-Calculus. *Information and Computation*, 91:189–204, 1991.
6. Peter Dybjer. Inductive Sets and Families in Martin-Löf’s Type Theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*. CUP, 1991.
7. Eduardo Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs, ’94*, volume 996 of *LNCS*, pages 39–59. Springer-Verlag, 1994.

8. Healfdene Goguen. *A Typed Operational Semantics for Type Theory*. PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1994. Available from <http://www.lfcs.informatics.ed.ac.uk/reports/94/ECS-LFCS-94-304/>.
9. Martin Hofmann and Thomas Streicher. A groupoid model refutes uniqueness of identity proofs. In *Proc. Ninth Annual Symposium on Logic in Computer Science (LICS) (Paris, France)*, pages 208–212. IEEE Computer Society Press, 1994.
10. Jean-Pierre Jouannaud and Claude Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 257–321. MIT Press, 1991.
11. Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, 1994.
12. Zhaohui Luo and Robert Pollack. LEGO Proof Development System: User’s Manual. Technical Report ECS-LFCS-92-211, Laboratory for Foundations of Computer Science, University of Edinburgh, 1992.
13. Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, LNCS 806. Springer-Verlag, 1994. Selected papers from the Int. Workshop TYPES ’93, Nijmegen, May 1993.
14. Per Martin-Löf. A theory of types. Manuscript, 1971.
15. Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999. Available from <http://www.lfcs.informatics.ed.ac.uk/reports/00/ECS-LFCS-00-419/>.
16. Conor McBride. Elimination with a Motive. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *Types for Proofs and Programs (Proceedings of the International Workshop, TYPES’00)*, volume 2277 of LNCS. Springer-Verlag, 2002.
17. Conor McBride, Healfdene Goguen, and James McKinna. A few constructions on constructors. In Jean-Christophe Filliâtre, Christine Paulin, and Benjamin Werner, editors, *Types for Proofs and Programs, Paris, 2004*, LNCS. Springer-Verlag, 2004.
18. Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1), 2004.
19. Fred McBride. *Computer Aided Manipulation of Symbols*. PhD thesis, Queen’s University of Belfast, 1970.
20. Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf’s type theory: an introduction*. Oxford University Press, 1990.
21. Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. In D. Basin and B. Wolff, editors, *Proceedings of the Theorem Proving in Higher Order Logics 16th International Conference*, volume 2758 of LNCS, pages 120–135, Rome, Italy, September 2003. Springer.
22. Thomas Streicher. Investigations into intensional type theory. Habilitation Thesis, Ludwig Maximilian Universität, 1993.
23. Daria Walukiewicz-Chrzaszcz. Termination of rewriting in the calculus of constructions. *J. Funct. Program.*, 13(2):339–414, 2003.