Interactive Program Derivation

Martin David Coen

St. John's College

A dissertation submitted for the degree of
Doctor of Philosophy in the University of Cambridge

March 1992

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration. This dissertation is not substantially the same as any I have submitted for a degree, diploma or other qualification at any other university.

Acknowledgements

I should like to thank my research supervisor, Dr Lawrence Paulson, for the freedom he allowed me in my research, and the interest he took in my work. His patient consideration of my ideas and the influence of his own research have been of great value throughout the development of this work. Thanks is also due to Tobias Nipkow and Andy Pitts for many valuable discussions about various aspects of the work.

This dissertation has benefitted from comments made on earlier drafts by Gavin Bierman, Marcus Moore, Valeria de Paiva and Clive Tong. Tom Melham also kindly read a draft and made helpful suggestions, both about the content of the dissertation and its layout. Thanks are due to Graham Titmus for his stalwart efforts in keeping the machines in the laboratory running, and to Margaret Levitt for keeping the bureaucratic wheels turning. I should also like to thank the Laboratory for the Foundations of Computer Science for the time I spent in Edinburgh during my second year.

Financial support for this work came from several sources. The main part of the work was supported by a studentship from the Science and Engineering Research Council. I am grateful to the ESPRIT Basic Research Action 3245 'Logical Frameworks' for equipment and funding to attend conferences, as well as support in the final months. St John's College also helped with funding to attend conferences.

Last, but by no means least, I should like to thank my family and Deborah Lamb for their continued support during the last few years.

Summary

As computer programs are increasingly used in safety critical applications, program correctness is becoming more important; as the size and complexity of programs increases, the traditional approach of testing is becoming inadequate. Proving the correctness of programs written in imperative languages is awkward; functional programming languages, however, offer more hope. Their logical structure is cleaner, and it is practical to reason about terminating functional programs in an internal logic.

This dissertation describes the development of a logical theory called CCL for reasoning about the correctness of terminating functional programs, its implementation using the theorem prover Isabelle, and its use in proving formal correctness. The theory draws both from Martin-Löf's work in type theory and Manna and Waldinger's work in program synthesis. It is based on classical first-order logic, and it contains terms that represent classes of behaviourally equivalent programs, types that denote sets of terminating programs and well-founded orderings. Well-founded induction is used to reason about general recursion in a natural way and to separate conditions for termination from those for correctness.

The theory is implemented using the generic theorem prover Isabelle, which allows correctness proofs to be checked by machine and partially automated using tactics. In particular, tactics for type checking use the structure of programs to direct proofs. Type checking allows both the verification and derivation of programs, reducing specifications of correctness to sets of correctness conditions. These conditions can be proved in typed first-order logic, using well-known techniques of reasoning by induction and rewriting, and then lifted up to CCL. Examples of program termination are asserted and proved, using simple types. Behavioural specifications are expressed using dependent types, and the correctness of programs asserted and then proved. As a non-trivial example, a unification algorithm is specified and proved correct by machine.

The work in this dissertation clearly shows how a classical theory can be used to reason about program correctness, how general recursion can be reasoned about, and how programs can direct proofs of correctness.

Contents

List of Figures

Chapter 1

Introduction

As programming has moved from machine code to higher level languages removed from hardware, there has been a change in the understanding of the craft. The creation of high level languages with reasonably clean logical structure has led to some discipline in program development. Dijkstra illustrated this change in attitude by the change in terminology from 'computer science' to 'computing science' [16, page 210]. With the continuing growth in the size and complexity of programs and their use in safety critical applications, assurance of correctness is becoming increasingly important. The traditional approach of testing is inadequate. Formal reasoning now provides a more viable approach, as the semantics of programming languages has become clearer and theorem proving technology more developed. This dissertation introduces a formal system for reasoning about the correctness of functional programs. More importantly, it describes the implementation of the system and its use in proving some significant examples.

1.1 Formal Program Derivation

A *computational logic* is a formal system for reasoning about the behaviour of programs in a *target* programming language. This encompasses both *external* logics such as those of Floyd and Hoare [2], which provide logical comments external to the program, and *internal* logics such as Martin-Löf's Type Theory(MLTT) [40], in which programs are represented as terms in the logic. A *specification* in a computational logic defines a class of programs satisfying the desired input/output behaviour. Proving the *correctness* of a program with respect to a specification is by proving membership in the defined class. A computational logic should allow specifications of varying flexibility, from simply requiring that a program terminates to completely defining its behaviour for all allowed inputs.

Before considering computational logics further, it is important to note their limitations. Formal methods are not a panacea. Formal correctness must rely on a formal specification. For large or complex applications, a formal specification is likely to contain discrepancies with the intended behaviour. Proving the correctness of a program with respect to a flawed specification cannot guarantee the intended behaviour. Furthermore, proving correctness in a computational logic assures the specified behaviour only with respect to a particular abstract model of computation in the target language. Correctness of an actual program must rely on the implementation (compiler, hardware, etc.) being faithful to this model.

That said, the exercise of formal reasoning is not futile. By allowing abstract specification of program behaviour, computational logics permit much clearer and more concise specifications than are possible in the target programming language itself, narrowing the gap between a designer's intentions and the formal description. An abstract model of computation can be chosen to be an idealisation of an implementation (for the logic in Chapter 3, a Natural Semantics [23] is used). Cohn presents a good discussion of these issues in the context of hardware verification [11].

The purpose of formalising program correctness in a computational logic is to exploit machine-based theorem provers. *Proof checking* by machine has greatly advanced over the past few years. Systems now exist in which it is possible to carry out proofs interactively using tactics to perform some parts automatically (e.g. HOL [20], Isabelle [54] and NuPrl [12]). Using this technology, many areas of mathematics have been formally developed (e.g. ZF set theory and the real numbers). Whether this is a useful exercise in itself is another matter, but proving the correctness of programs is one area of mathematics in which there are clear advantages to a formal approach. Correctness proofs tend to be straightforward but long and full of detail, which makes them tedious and prone to error when done by hand. Machine-based theorem provers can keep track of the tedious detail in proofs and automate the simpler parts.

1.2 Previous Work

Early work on formal program development, as opposed to the informal methodologies proposed by Dijkstra [16] and Gries [22], was done by Burstall and Darlington [7] using *program transformation*. A clear but inefficient program is used to specify a problem. It is transformed into a less clear but more efficient program by a series of steps, each of which is guaranteed to preserve the program's behaviour. Using a program as specification defines the class of programs with exactly the same input/output behaviour. Though much can be done within this framework (see Bird [5, 6]), it is too concrete. For example, specifying a compiler by presenting one possible implementation fixes the code generated for each source program, so prohibiting transformation to a compiler that generates better code. Specifications must be more abstract, describing the task required rather than a particular implementation.

Floyd-Hoare techniques have been used to formally *verify* that imperative programs satisfy their specifications; logics such as LCF have been used for verifying functional programs [51]. More recently, formal methods have been devised for the *synthesis* of programs in high level languages hand-in-hand with the development of their proofs of correctness. Manna and Waldinger [34] have used tableau-based theorem proving to synthesise general recursive, functional programs from first-order specifications.

Functional programming languages offer a relatively clean logical structure, satisfying Leibniz's Law (the substitutivity of equals for equals), in contrast to imperative languages in which the simple structure is destroyed by constructions such as assignment and aliasing. Moreover, correctness proofs for functional programs closely follow the structure of the algorithms themselves; an algorithm can be used as a template to direct its proof of correctness. Both verification and synthesis can be directed in this way. In verification, the algorithm fixes the template from the outset; whereas in synthesis, the template (and hence the algorithm) is incrementally instantiated as the proof progresses. The two styles may be freely mixed in one proof, verifying an algorithm where it is given and synthesising it elsewhere. This is a promising approach to program correctness; it is much easier to write programs than to prove their correctness.

In an intuitionistic framework, there is an identification between proofs and programs known as the Curry-Howard isomorphism. The insights that arise from formalising this isomorphism in type theories have led to a plethora of computational logics: including work using MLTT [8], NuPRL [12] and the Calculus of Constructions [45]. In particular, Paulin [50] and Hayashi [24] consider extracting a functional program from an intuitionistic proof that its specification can be met. Dybjer [17] has suggested using Aczel's theory LTC, an untyped framework, as a computational logic.

1.3 Kinds of Computational Logic

Two possible bases for a computational logic are denotational and operational semantics.

- In a *denotational logic*, each construction in the target programming language is given meaning by the term it denotes in the logic. Properties of actual programs can be inferred provided there is a good fit between the mathematical meaning of terms and their computational behaviour (at least computational adequacy [43]). Giving meaning to non-terminating programs requires partial functions or the addition of an element representing "undefined" to every type. Domain theory takes the second approach: a *domain* is a complete partial ordering of a set with a bottom element $\perp$. Programs are continuous functions between domains. The computational logic LCF [21, 53, 10] is based on domain theory. Although interesting non-terminating programs do exist, the inclusion of an extra case ($\perp$) in proofs makes logics such as LCF unnecessarily awkward for reasoning about terminating programs [51]. If only terminating programs are considered, then set theory can be used. Programs denote total functions, but require annotations. For example, λ-abstraction is typed, and the operator for general recursion includes a well-founded relation. In this context, data types can be constructed, and general recursive programs can be denoted using well-founded recursion.

- In an *operational logic*, programs are represented by abstract syntax trees. An evaluation relation is defined between programs to capture the operational semantics, and this is used to define an equivalence relation. Two programs are equivalent if they behave in the same way in all contexts (observational congruence [43]). Properties of programs (including correctness) are expressed with respect to this equivalence. This is the approach adopted in Chapter 3.

Intuitionistic logics provide an appealing framework for computational reasoning. Proofs have computational content; if a proposition is regarded as a program specification, then its proof contains an algorithm that meets this specification. Complications arise when useful forms of inductive types and recursion are considered. Much work is being done in the development of intuitionistic computational logics [40, 4, 25, 33]. Although intuitionistic logic is not used here, it has had much influence on this work (see §2.2). But the intuitive understanding of logic, at least amongst programmers, is and is likely to remain classical. Unless the chosen model of computation prohibits a classical framework, there is no reason to insist that a computational logic be intuitionistic.

As a computational logic, typed first-order logic[1] is unsatisfactory, because new types cannot be defined within the logic but require additional axioms, and only simply typed

[1]This is often called 'sorted first-order logic'. But the word sort has a specific meaning in the context of the theorem prover Isabelle, so to avoid any confusion it is not used here.

total functions may be considered. But it is still a useful framework for reasoning within particular domains. For example, first-order logic, extended with a type of natural numbers *nat* together with axioms stating the behaviour of the constructors 0 and S and the primitive recursive operator *nrec*, is a suitable theory in which to define the arithmetic functions $+$ and $\times$, and prove facts about them. Several types can be present in one theory, for example booleans and natural numbers; the well-formedness of formulae ensures that each set of axioms applies only to terms of the appropriate type. Types can also be polymorphic, for example the type of *List(*α*)* of lists of α, where α is a variable ranging over types. For a given programming problem, data types and simply typed functions over them can be axiomatised in typed first-order logic, and facts derived within this framework (see §4.5).

Gordon's formulation [20] of Church's higher-order logic [9] allows the construction of new data types [42]. It too uses only simple types, not depending on terms. Simple types work well as *partial* specifications, not requiring termination; the host of strongly typed functional languages bears witness to this. But for *total* specifications requiring termination, simple types are insufficient. For example, a function for subtractive division can be defined in ML by

```
fun div(n,d) = if (n<d) then 0 else div(n-d,d) + 1;
```

with the ML type `nat*nat -> nat`. Termination can be proved for all $n \in Nat$ and all $d \in \{x \in Nat \mid x \neq 0\}$. But the function cannot be typed as a total function $Nat * Nat \rightarrow Nat$, it needs a further condition on the input $Nat * \{x{:}Nat, x \neq 0\} \rightarrow Nat$. A simply typed version of this function must include an arbitrary value for `div(n,0)`.

```
fun div(n,0) = <dummy>
  | div(n,d) = if (n<d) then 0 else div(n-d,d) + 1;
```

1.4 The Aims of this Work

This dissertation considers the implementation and use of a computational logic for proving the correctness of terminating functional programs. The target language is a core functional programming language, including general recursion, but excluding non-determinism, concurrency and imperative features.

The logic CCL (Classical Computational Logic) is developed and implemented using the theorem prover Isabelle with a collection of tactics to support reasoning about program correctness. The intention is not that CCL is to be the final word in computational logics—it clearly isn't—but rather that the development and use of CCL may shed some light on what makes a computational logic effective. It is hoped that this dissertation will convince the reader that

- classical logics are suitable for reasoning about program correctness,
- the economy of considering only total functions need not preclude general recursion, and
- using programs as templates to direct correctness proofs is an effective approach to formal program development.

1.5 Outline of the Dissertation

- Chapter 2 describes the work that has had the greatest influence on this dissertation. In particular, Mana and Waldinger's use of well-founded induction and deductive tableaux for reasoning about general recursion, the foundational work in Martin-Löf's type theories on the logical nature of types for functional languages, and Paulin's work on extracting programs from constructive proofs.

- Chapter 3 introduces the computational logic CCL. From a formulation of the target programming language, the constants of CCL are defined and its rules derived. Properties of CCL are proved that justify its use as a computational logic.

- Chapter 4 describes the implementation of CCL using the generic theorem prover Isabelle. Tactics are developed for type checking, rewriting and other more specific aspects of program correctness. First-order logic, extended with computational types, is used to prove many of the lemmas necessary for correctness. CCL acts as a meta-logic in which to interpret first-order logic.

- Chapter 5 describes how the implementation of CCL is used to prove program correctness. An extended example is considered: an algorithm for unification is introduced, formally specified within CCL and derived using the implementations of CCL and first-order logic.

- Chapter 6 summarises the main contributions of this dissertation and describes some areas of further research suggested by the work.

- Appendix A introduces well-founded induction and constructions for well-founded relations.

Chapter 2

Review of Other Work

In Chapter 3, a computational logic is developed for a target programming language with a particular notion of evaluation. Although the motivations are quite different, the major inspiration for this comes from Martin-Löf's type theoretic approach to logic [39, 40], and the subsequent work in Computing Science based on this. But in contrast to type theory, general recursion is considered; Manna and Waldinger's work on program synthesis clearly shows the merits of this [34].

§2.1 describes Manna and Waldinger's deductive tableau system, and its use in program synthesis. §2.2 introduces the basic ideas behind Type Theory, and considers how they lead to a computational logic. §2.3 outlines the work of Paulin [50], which has similar aims to mine but takes a somewhat different approach. Finally, §2.4 briefly describes the relevance of this work to the approach taken in the dissertation.

2.1 Deductive Tableaux

For many years, Manna and Waldinger have been proponents of program synthesis as a theorem proving task. They developed a system of deductive tableaux in which proving that a specification can be met synthesises a functional program.

Their deductive system is based on the notion of a *tableau*, which consists of a collection of rows each with three columns. Each row contains a single sentence of predicate logic, called an *assertion* if it appears in the first column and a *goal* if it appears in the second. The third column may sometimes contain an *output expression*, which records the program fragment that has been constructed at a particular stage of the proof, but has no bearing on the proof itself. Each row in a tableau has either the form

assertions	goals	output
$A_i(\overline{a}, \overline{x})$		t_i

or

	$G_i(\overline{a}, \overline{x})$	t_i

where $\overline{a}$ denotes all the constants and $\overline{x}$ all the free variables in a goal or assertion. As a convention, write $a, b, c, f, g \ldots$ for constants and $x, y, z \ldots$ for variables. The variables in a row are "dummies", which can be systematically renamed without changing the meaning of the row. The meaning of a tableau with assertions $A_i(\overline{a}, \overline{x})$ and goals $G_i(\overline{a}, \overline{x})$ is given by the following associated sentence of predicate logic.

$$\left(\bigwedge_i \forall \overline{x}.\ A_i(\overline{a},\overline{x})\right) \quad \supset \quad \left(\bigvee_i \ \exists \overline{x}.\ G_i(\overline{a},\overline{x})\right)$$

A tableau is therefore *valid* iff under all interpretations at least one of the assertions is false or at least one of the goals is true. Clearly, the distinction between assertions and goals is superfluous; a sentence that appears as an assertion (goal) could instead appear in its negation as a goal (assertion). The distinction is only for clarity in the deductions.

A program is specified by an input/output relation. Given an input a such that $P(a)$ holds, the program should output a value b such that the relation $R(a,b)$ holds. Manna and Waldinger write this as

$$\begin{array}{lll} f(a) & \Leftarrow & \textit{find } z \textit{ such that } \ R(a,z), \\ & & \textit{where } \ P(a) \end{array}$$

which corresponds to the theorem

$$\forall a.\ P(a) \ \supset \ \exists z.\ R(a,z)$$

Synthesis begins with a tableau representing this theorem.

assertions	goals	output $f(a)$
$P(a)$		
	$R(a,z)$	z

At each step in the deduction, rows are added to the tableau. Never is there a requirement to delete a row (though it may help proof search to do so). A deduction terminates when a *terminal* row is reached (i.e. one which makes the tableau trivially valid), either

	true	t

or

false		t

The synthesised program is then

$$f(a) \ \Leftarrow \ t$$

A deduction step uses a rule from one of the following categories: splitting, transformation, non-clausal resolution and recursive calls. For brevity, only the propositional part of the deduction system is described, though it has been developed for predicate logic [37, Chapter 11].

Splitting rules allow assertions and goals to be decomposed into their logical components. They are simple consequences of the meaning of tableaux. There are rules *andsplit*, *orsplit* and *ifsplit* corresponding to the sequent rules $\wedge$-left, $\vee$-right and $\supset$-right. For example, the rule *andsplit* is written as

$F \wedge G$		t
F		t
G		t

meaning that if the rows above the double line are present in the tableau, then those below the double line may be added. There are no rules corresponding to the sequent rules $\wedge$-right, $\vee$-left and $\supset$-left; instead, transformation rules are used in particular instances.

Transformation rules allow one sentence (either an assertion or a goal) to be derived from another. The transformation rule

$$r \Rightarrow s \ \ \mathit{if} \ \ P$$

means that r is a logically equivalent sentence to s, or that r is a term equal to s, provided P holds. If r can be unified with a subexpression of F, and θ is the most general unifier, then for this rewrite rule the following deduction rules allow substitution in assertions and goals respectively.

F		t
$P\theta \supset F\theta[s\theta/r\theta]$		$t\theta$

	F	t
	$P\theta \wedge F\theta[s\theta/r\theta]$	$t\theta$

where $t\theta$ denotes the application of the substitution θ to the expression t, and $t[y/x]$ denotes the substitution of y for all free occurrences of x in t. Transformation rules are more than simple rewriting; they allow arbitrary procedures. For example, the rule

$$f(x) = a \quad \Rightarrow \quad x = e \quad \mathit{if} \ \ P$$

represents solving the equation $f(x) = a$ for x under a condition P. They can also be used for simple inference. The rewrite rule

$$f(a) = f(b) \quad \Rightarrow \quad \mathit{true} \quad \mathit{if} \ \ a = b$$

reduces a goal $f(a) = f(b)$ to the goal $a = b$. If the side condition of a rule is vacuous (*true*), then it is omitted.

$$P \wedge \mathit{true} \quad \Rightarrow \quad P$$

reduces any subexpression that matches the left-hand side.

Non-Clausal Resolution allows subsentences of two rows to be unified, and then eliminated by classical case analysis. This generalises the conventional resolution of Robinson [58], allowing greater freedom in how it can be used—a problem for automation. Consider two rows, containing assertions F and G. If θ is the most general unifier of subsentences of F and G (P_F and P_G respectively), then

F		t_1
G		t_2
$F\theta[true/P_F\theta] \lor G\theta[false/P_G\theta]$		*if* P_F *then* $t_1\theta$ *else* $t_2\theta$

If neither initial row has an output term, then none is created for the new row; if only one of the initial rows has an output term t, then the output term for the new row is $t\theta$. Other rules can be deduced for the cases when F and G are an assertion and a goal, a goal and an assertion, and two goals. The case for a goal and an assertion is also shown, because of its use with the recursion hypothesis below.

	F	t_1
G		t_2
	$F\theta[true/P_F\theta] \land \neg G\theta[false/P_G\theta]$	*if* P_F *then* $t_1\theta$ *else* $t_2\theta$

This should be read with the same proviso on the output expression as above.

Recursive Calls are the most interesting feature of this work. The assertion

$(u \prec_w a \land P(u)) \supset R(u, f(u))$		

can always be added to a tableau as a general induction hypothesis. It states that if there exists some u such that $u \prec_w a$ holds (where a is the formal parameter in the original specification, and w is an, as yet, undetermined well-founded relation) and $P(u)$ holds (where P is the original input condition), then $R(u, f(u))$ holds (where f is the name of the function being synthesised, and $f(u)$ is a recursive call). Using resolution (for a goal and an assertion) and some transformations, this assertion can be used with the row

	$R(s, z)$	$t(z)$

to deduce the new row

	$s \prec_w a \land P(s)$	$t(f(s))$

which introduces the recursive call $f(s)$ into the output expression. The first conjunct ($s \prec_w a$) is the termination condition. This is an effective way to handle recursion for two reasons. First, the induction principle is well-founded induction (see §A), which allows the synthesis of any form of recursion that terminates. Second, the choice of well-founded relation w, and hence the form of recursion, need not be fixed at the time the recursive call is considered. It is possible to solve the condition on partial correctness (i.e. the conjunct $P(s)$) before considering the termination condition. Each recursive call in a synthesised program produces a goal for termination. If these goals are satisfiable, then at the end of the proof a relation can be chosen and the goals proved.

As an example, consider the inductive step used in synthesising a program to find the integer quotient of two numbers from the following specification.

$$\begin{aligned} div(n,d) \Leftarrow \quad & \textit{find } q \textit{ such that } \exists r.\ r < d \ \land\ r \geq 0 \ \land\ n = q \times d + r, \\ & \textit{where } n \geq 0 \ \land\ d > 0 \end{aligned}$$

For simplicity, type conditions are omitted from the specification and from the following tableaux. They would appear in the logic as predicates, for example *integer*(n). The initial tableau is

assertions	goals	output $div(n,d)$
$n \geq 0 \ \wedge \ d > 0$		
	$\exists r.\, r < d \ \wedge \ r \geq 0 \wedge$ $n = q \times d + r$	q

After some arithmetic reasoning, the following row is added by transformation.

	$\exists r.\, r < d \ \wedge \ r \geq 0 \wedge$ $n - d = x \times d + r$	$x + 1$

As this goal is an instance of the initial goal, it is appropriate to add an instance of the induction hypothesis, in which R has the form of the initial assertion and P has the form of the initial goal.

$\langle u, v\rangle \prec_w \langle n, d\rangle \ \supset \ u \geq 0 \wedge v > 0 \ \supset$ $\exists r.\, r < v \ \wedge \ r \geq 0 \ \wedge \ u = div(u,v) \times v + r$		

Applying resolution between the last two rows followed by some transformations gives

	$\langle n-d, d\rangle \prec_w \langle n, d\rangle \ \wedge$ $n - d \geq 0 \ \wedge \ d > 0$	$div(n-d,d) + 1$

where the unifying substitution is

$$\theta \ = \ \{n - d/u, \ d/v, \ div(n-d,d)/x\}$$

At this point, the relation w is uninstantiated. It is chosen so that the termination condition can be solved. Here, the less-than relation on the first component of the pairs is sufficient. The derivation is completed by further transformations and some case analysis by resolution, synthesising the program

$$div(n,d) \ \Leftarrow \ \mathit{if} \ \ n < d \ \ \mathit{then} \ \ 0 \ \ \mathit{else} \ \ div(n-d,d) + 1$$

Some guidance is provided by a *polarity strategy*, which restricts the use of resolution by insisting that a positive occurrence of one subsentence is used only with a negative occurrence of the other; and by a *recurrence strategy*, which determines when an induction hypothesis should be introduced.

Using this system by hand, programs have been synthesised from specifications for subtractive division [34], various forms of sorting [63], integer square root [36] and unification [35]. Some of the examples, notably unification, use forms of recursion that could not be easily handled by the primitive recursion rules of type theory. More recently, the deductive tableau system has been implemented by machine.

2.2 Type Theory

Many insights into formally reasoning about functional programs come from examining the relationship between programs and proofs in intuitionistic logic. Below, a type theory is introduced that closely resembles some versions of Martin Löf's Type Theory, and its application to program derivation is considered.

A proposition is intuitionistically valid iff a proof of it can be given. The meanings of the logical connectives are given in terms of proofs. Heyting [26] gave an informal explanation along the following lines.

a proof of	consists of
$\perp$	–
$A \wedge B$	a proof of A and a proof of B
$A \vee B$	a proof of A or a proof of B together with an indication of which it is
$A \supset B$	an operation that when applied to a proof of A yields a proof of B
$\forall x.\ B(x)$	an operation that when applied to an object a yields a proof of $B(a)$
$\exists x.\ B(x)$	an object a together with a proof of $B(a)$

Such a concrete notion of proof can be made more explicit by introducing a language of *proof terms.*

a proof of	has the form of
$\perp$	–
$A \wedge B$	a pair $\langle a, b\rangle$, where a is a proof of A and b is a proof of B
$A \vee B$	either $\mathsf{inl}(a)$, where a is a proof of A, or $\mathsf{inr}(b)$, where b is a proof of B
$A \supset B$	a function $\mathsf{lam}\ x.b(x)$, where $b(a)$ is a proof of B provided that a is a proof of A
$\forall x.\ B(x)$	a function $\mathsf{lam}\ x.b(x)$, where $b(a)$ is a proof of $B(a)$ for an object a
$\exists x.\ B(x)$	a pair $\langle a, b\rangle$, where a is an object and b is a proof of $B(a)$

As the meaning of a formula is given by its proof, a proposition can be identified with the type of all its proofs. This gives a correspondence between judgements of the form "a inhabits type A", written $a : A$, and judgements of the form "a is a proof of A" known as the Curry-Howard isomorphism.

There is a correspondence between simple type theory and propositional logic, shown in the following table.

Proposition	Type	Proof Term	
		canonical	non-canonical
$\bot$	0	–	
$A \wedge B$	$A \times B$	$\langle a, b\rangle$	$\mathsf{split}(p, f)$
$A \vee B$	$A + B$	$\mathsf{inl}(a)$,$\mathsf{inr}(b)$	$\mathsf{when}(p, f, g)$
$A \supset B$	$A \rightarrow B$	$\mathsf{lam}\ x.b(x)$	$\mathsf{apply}(f, a)$

A proof of the proposition $A \supset A \vee B$ inhabits the type $A \rightarrow A + B$, for example the term $\mathsf{lam}\ x.\mathsf{inl}(x)$. Quantification requires indexed families of types, that is types of the form $B(x)$ indexed by an object x of type A.

Proposition	Type	Proof Term	
		canonical	non-canonical
$\forall x{\in}A.\ B(x)$	$\mathsf{\Pi}x{:}A.B(x)$	$\mathsf{lam}\ x.b(x)$	$\mathsf{apply}(f, a)$
$\exists x{\in}A.\ B(x)$	$\Sigma x{:}A.B(x)$	$\langle a, b\rangle$	$\mathsf{split}(p, f)$

The type $\mathsf{\Pi}x{:}A.B(x)$ is the product of a family of types, each object of which is a function that applied to an object a of type A, yields an object of type $B(a)$. The type $\Sigma x{:}A.B(x)$ is the sum of a family of types, each object of which is a pair $\langle a, b\rangle$, where a is an object of type A and b is an object of type $B(a)$.

The non-canonical terms select information from proofs. Their meanings are given by the following implicit operational semantics.

- split is an operation that takes a term p and an operation f. If p yields the canonical term $\langle a, b\rangle$ then split yields the result of $f(a, b)$, i.e. $\mathsf{split}(\langle a, b\rangle, f) \leadsto f(a, b)$.
- when is an operation that takes a term u and two operators f and g. If u yields the canonical term $\mathsf{inl}(a)$ then when yields the result of $f(a)$, and if u yields the canonical term $\mathsf{inr}(b)$ then when yields the result of $g(b)$, i.e. $\mathsf{when}(\mathsf{inl}(a), f, g) \leadsto f(a)$ and $\mathsf{when}(\mathsf{inr}(b), f, g) \leadsto g(b)$.
- apply is an operator that takes terms f and a. If f yields the canonical term $\mathsf{lam}\ x.b(x)$ then apply yields the result of $b(a)$, i.e. $\mathsf{apply}(\mathsf{lam}\ x.b(x), a) \leadsto b(a)$.

Under the interpretation of propositions-as-types, the proof rules of natural deduction correspond to rules of type inference. For example, the natural deduction rule $\wedge I$ corresponds to the type inference rule for pairing,

$$\frac{A \qquad B}{A \wedge B} \qquad\qquad \frac{a : A \qquad b : B}{\langle a, b\rangle : A \times B}$$

and the rule $\wedge E$ corresponds to the type inference rule for split.

$$\frac{A \wedge B \qquad \begin{array}{c}\left[\, A \,;\; B \,\right] \\ C\end{array}}{C} \qquad\qquad \frac{p : A \times B \qquad \begin{array}{c}\left[\, x : A \,;\; y : B \,\right]_{x,y} \\ c(x,y) : C\end{array}}{\mathsf{split}(p, c) : C}$$

The notation $[\;]_{x,y}$ binds the eigenvariables x and y in the inference $x : A \;\wedge\; y : B \implies c(x, y) : C$. It is described more fully in §3.1. There are also equality rules in type theory, reflecting the implicit operational semantics of the non-canonical terms.[1] For example

$$\frac{a : A \qquad b : B \qquad \begin{array}{c}\left[\, x : A \,;\; y : B \,\right]_{x,y} \\ c(x,y) : C\end{array}}{\mathsf{split}(\langle a, b\rangle, c) = c(a, b) : C}$$

For type theory to provide a useful framework for *proof checking*, it must be possible to check that a proof a actually proves A; the judgement $a : A$ must be decidable when a is instantiated. Clearly, if a is unknown, then finding a proof of A is in general undecidable for anything beyond propositional reasoning. That $a : A$ is decidable is a consequence of term reduction being Church-Rosser and strongly normalising. Type theory admits an interpretation of logic in which propositions correspond to types, proofs to terms and proof normalisation (cut elimination) to term reduction.

There is also an interpretation of type theory as a programming language. Terms and types represent the usual data structures; term reduction gives an operational semantics for the terms. The essential difference between this and a strongly typed functional language such as ML is that all terms of the type theory represent terminating programs.

It is natural to extend the purely logical type theory with inductive reasoning, either by encoding new inductive schemes into a general well-ordered type [49, Chapters 15 and 16], or by considering the theory as an open system to which axiomatisations of new types may be added [3]. For example, the type of natural numbers Nat has introduction rules for zero (zero) and successor (succ),

$$\mathsf{zero} : \mathsf{Nat} \qquad\qquad \frac{n : \mathsf{Nat}}{\mathsf{succ}(n) : \mathsf{Nat}}$$

and an elimination rule, which allows structural induction (in this case just mathematical induction).

$$\frac{n : \mathsf{Nat} \qquad b : B(\mathsf{zero}) \qquad \begin{array}{c}\left[\, x : \mathsf{Nat} \,;\; y : B(x) \,\right]_{x,y} \\ c(x,y) : B(\mathsf{succ}(x))\end{array}}{\mathsf{nrec}(n, b, c) : B(n)}$$

The non-canonical constant has reductions

$$\mathsf{nrec}(\mathsf{zero}, b, c) \rightsquigarrow b \qquad\qquad \mathsf{nrec}(\mathsf{succ}(n), b, c) \rightsquigarrow c(n, \mathsf{nrec}(n, b, c))$$

corresponding to primitive recursion over Nat.

[1] These correspond to the reduction rules of Prawitz for natural deduction proofs [57].

The judgement $a : A$ can be read in three ways

- a inhabits type A,
- a is a proof of proposition A, or
- a is a program satisfying specification A.

An algorithm can be derived as the constructive proof of the proposition

$$\forall x \in A.\ \exists y \in B.\ C(x, y)$$

where $C(x, y)$ specifies the relation between input and output. A term inhabiting the corresponding type has the form

$$\mathsf{lam}\ x. \langle b, p\rangle\ :\ \Pi x{:}A.\Sigma y{:}B.C(x, y)$$

which is a function from an object $a : A$ to a pair $\langle b, p\rangle$ in which b is the desired result and p is a proof that $C(a, b)$ holds.

Type theory is a good approach to formal reasoning about functional programs, as it reveals the similarities between programs and proofs. Branching in a program is reasoned about using case analysis, and recursion using induction. For the type theory described above, the two are combined; primitive recursion in programs corresponds to structural induction in proofs. Moreover, there is a single rule for each program term-former, making it simple to direct a proof with a program template. For the theory presented above, this is a trivial observation since programs are identified with proofs. It becomes more interesting when a distinction is made between proofs and programs.

Dependent types allow the input/output behaviour of programs to be easily expressed in specifications. Assume a subtype type-former. For some program f, a predicate of the form

$$\forall x.\ P(x) \supset Q(x, f(x))$$

is expressed as

$$f : \Pi x{:}\{x{:}A, P(x)\}.\{y{:}B, Q(x, y)\}$$

Without dependent types, specifications become slightly longer and more clumsy,

$$f : \{u{:}\{x{:}A, P(x)\} \rightarrow B, \forall x{:}A.P(x) \supset Q(x, u(x))\}$$

which is awkward when they are used as hypotheses for recursive calls.

But there are complications to this approach. Terms are used on the one hand to represent proofs, and on the other to represent programs. Giving two distinct roles to one object is not ideal; a proof object necessarily contains a complete justification of a proposition, whereas a program need contain only the algorithm for computing the result. For a programming language, reduction should be determinant and terminating under a particular strategy. The stronger conditions of Church-Rosser and strong normalisation, necessary for a term to act as a proof object, are too restrictive. Strong normalisation leads to a restricted form of recursion—primitive recursion. Although primitive recursion allows all feasible computations to be expressed, for example the encoding of Ackermann's function as a primitive recursive function of higher type (a functional) by Nordström [47], it does not provide a style that is at all practical for programming (see §2.4).

For program derivation, type theory needs to be adapted to

- remove computationally uninteresting information from program terms, and
- provide general forms of recursion.

Computationally uninteresting information can be discarded by extracting programs from completed proofs. This is the basis of the work done in the Calculus of Constructions (see §2.3). Alternatively, subtyping can be used to discard the redundant information as a term is constructed.

A subset type-former can be introduced with the following introduction and elimination rules.

$$\frac{a : A \qquad B(a)\ \mathit{true}}{a : \{x{:}A, B(x)\}}$$

$$\frac{a : \{x{:}A, B(x)\}}{a : A} \qquad\qquad \frac{a : \{x{:}A, B(x)\}}{B(a)\ \mathit{true}}$$

But this does not easily fit together with the notion of propositions-as-types. For the subset construct to work properly, a distinction must be made between propositions and types [59].

Paulson [52] describes an attempt to handle recursion schemes other than primitive recursion by describing well-founded relations in type theory. Nordström [48] presents a notion of elements of A *accessible* through a relation $\prec$, $Acc(A, \prec)$. The relation $\prec$ is well-founded with respect to A if $Acc(A, \prec)$ is equal to A. This provides a rule of well-founded induction, with a proof term $\mathsf{rec}(e, p)$ in which e is the body of the recursion and p is the initial value.

Despite its limitations and the awkward style of reasoning, Martin Löf's Type Theory has been used for program derivation. In particular, Chisholm describes a machine checked derivation of a simple parsing algorithm [8].

2.3 Calculus of Constructions

Paulin developed a variant of the Calculus of Constructions [13] in which programs of F_ω may be extracted from proofs [50].

F_ω is a higher-order extension of the simply typed lambda calculus [19]. As well as *terms* and *types*, there is an additional level of *orders*. Types that inhabit the order *Data* are called *data types*. Terms that inhabit data types are called *programs*. All well-typed terms are strongly normalising. The usual data types are definable, but with iteration in place of primitive recursion. For example, instead of the primitive recursor for natural numbers, *nrec*, defined such that

$$\begin{array}{lll} \mathit{nrec}\ 0\ t\ u & \leadsto & t \\ \mathit{nrec}\ S(n)\ t\ u & \leadsto & u\ n\ (\mathit{nrec}\ n\ t\ u) \end{array}$$

there is an iterator, *nit*, defined such that

$$\begin{array}{lll} \mathit{nit}\ 0\ t\ u & \leadsto & t \\ \mathit{nit}\ S(n)\ t\ u & \leadsto & u\ (\mathit{nit}\ n\ t\ u) \end{array}$$

Although recursion may be encoded using iteration, this is very inefficient. For example, an iterative predecessor function on the natural numbers would decompose its argument to 0, and then construct a result with one fewer S nodes.

The Calculus of Constructions (CoC) is an impredicative higher-order lambda-calculus with dependent types, which has been used to formalise and check mathematical reasoning. F_ω is the part of CoC in which types are not dependent on terms. For reasoning about programs of F_ω in CoC, it is convenient to distinguish the program types (*Data*) from the propositions, and to further distinguish the *informative* propositions (*Spec*) from the *non-informative* ones (*Prop*). Informative propositions have computational content, and their proofs contain programs; non-informative propositions have only a 'logical' content, and their proofs should be discarded. Here too, propositions are separated from types, which allows subsets to be expressed, and additional axioms to be realised without producing an inconsistent environment.

Paulin uses a variant of CoC, the Calculus of Constructions with Realisations. The notion of *realisability* gives an internalisation of the computational meaning of proofs; it captures the idea of *extraction* in a precise and systematic way. For each proposition A, there is a realisability predicate $\lambda x\,.\,\mathcal{R}(A, x)$. If the judgement $t : A$ is derivable, then there is a proof of $\mathcal{R}(A, t')$ where t' is the program extracted from proof t. The extraction function, $t' = \mathcal{E}(t)$, is inductively defined over the structure of terms. It removes the non-informative parts of the term and all type dependencies to produce a program term of F_ω. The relation between extraction and realisability is illustrated by the following picture.

$$\begin{tikzcd}
 & \vdash t \in M \in \mathit{Spec} \arrow[dl, "\mathcal{E}"'] \arrow[dr, "\mathcal{R}"] & \\
\vdash \mathcal{E}(t) \in \mathcal{E}(M) \in \mathit{Data} & & \vdash u \in \mathcal{R}(M, \mathcal{E}(t))
\end{tikzcd}$$

Furthermore, if there exists a term u such that $\mathcal{R}(A, u)$ is derivable, then u is said to *realise* A and A is consistent in the theory. If the axiom $ax : A$ is added to the theory, then ax may appear in extracted terms. A reduction strategy is needed for ax, for which it is correct to replace ax by u in a program. This allows an axiom of well-founded induction to be added to the theory with a proof term that provides general recursion.

2.4 Comments

This dissertation is largely influenced by the work described above. But it should be noted that, despite initial appearances, classical logic is not incompatible with a computational logic, even when based on type theory. Taking a type theoretic approach to program correctness leads naturally to a system in which propositions and their proofs are distinguished from program types and their programs. Once this is done, propositional reasoning can be made classical (by adding a suitable axiom and a non-constructive proof term), whilst the program type system remains constructive. Moreover, though insufficient to represent propositions, sets can be used to represent program types, which allows the use of subsets, fixed points, etc. Similarly in CCL, the logical system is classical first-order logic (though not based on type theory), and the program type system is constructive (and types are just sets).

General recursion is essential for a programming language to be practical. Although primitive recursive functionals allow all feasible computations to be expressed, they do not always allow the most natural or efficient algorithm. Hoare's algorithm for quicksort is a familiar example that illustrates this point. Assume partition functions `les x xs` and `gts x xs` that return the elements of `xs` that are less-than or equal to `x` and greater than `x` respectively. Quicksort can be defined in ML by

```
fun qsort [] = []
  | qsort (x::xs) = (qsort (les x xs)) @ x::(qsort (gts x xs));
```

In ML, the primitive recursive encoding is

```
fun qsort l =
   let fun qsortx 0 l = []
         | qsortx n [] = []
         | qsortx n (x::xs) = (qsortx (n-1) (les x xs))
                              @ x::(qsortx (n-1) (gts x xs))
   in qsortx (length l) l end;
```

Clearly, this is unsatisfactory. Apart from its awkwardness, the encoding is inefficient; executing an application of the primitive recursive version of quicksort evaluates the measure function `length l` (a completely wasted computation) as well as sorting the list. Similarly, a primitive recursive function to find the next prime after some given integer must calculate an upper bound on the prime before searching for it.

For every terminating program, a well-founded relation can be found that orders the successive recursive calls. Well-founded induction can, therefore, be used as a general scheme for reasoning about recursion. A suitable proof rule is described by Paulson in the context of Martin-Löf's Type Theory [52]. Furthermore, with a single proof rule, the choice of recursion scheme (i.e. the instantiation of the well-founded relation) can be postponed to a later stage in a proof. For correctness proofs to be directed by algorithms this is essential; the form of recursion is not fixed at the outermost level of a function declaration, but at the points at which recursive calls are made. Manna and Waldinger's work on program synthesis uses well-founded induction in this way. By delaying the choice of well-founded relation until the end of a proof, partial correctness and termination can be separated—a technique practised since the advent of Floyd-Hoare logic.

Chapter 3

The Theory CCL

CCL is a classical logic for reasoning about terminating, general recursive, functional programs. The theory is presented as a natural deduction calculus and given meaning in a meta-language based on set theory, described in §3.1.

CCL contains an untyped, target programming language $\mathcal{L}$, described in §3.2. Programs in $\mathcal{L}$ are represented by their abstract syntax trees; and execution is defined by an *Evaluation Semantics*—a set of rules in the style of Natural Semantics [31, 44], a refinement of Structural Operational Semantics [56]. In §3.3, an observational equivalence is defined between programs (similar to applicative bisimulation [1] and observational congruence [43]). Two programs are equivalent in this sense if they evaluate to normal forms, which have the same outermost constructor and whose components are equivalent. Observational equivalence is proved to be a congruence relation over the term-formers of $\mathcal{L}$. It can, therefore, be used as an equality relation over equivalence classes of programs, and the term-formers can be lifted up to equivalence classes in the obvious way.

In §3.4, types are inductively defined as sets of equivalence classes of terminating programs. There is a universe of types; arbitrary subsets can be considered, and inductive types represented using a fixed point. Well-founded orderings, introduced in §3.5, are used to ensure termination over general schemes of recursion. They are built from a set of defined constructors, so that well-foundedness is just a well-formedness condition.

CCL includes the connectives of first-order logic. §3.6 describes the complete logic, and §3.7 presents a result about the strength of CCL. The theory is constructed from objects of the following syntactic categories.

- $\iota_=$ are equivalence classes of programs,
- τ are types of programs,
- ω are orderings well-founded over $\iota_=$, and
- o are formulae.

Formulae are those of classical first-order logic, with quantification over terms ($\iota_=$) and types (τ), together with the following predicates.

- $a : A$, the membership relation on types,
- $a = b$, an overloaded equality between terms and between types,
- $a \prec_R b$, the membership relation on well-founded orderings, and
- $Mono(f)$, which holds iff the function f from types to types is monotonic.

3.1 Meta-Theoretic Conventions

The metalanguage, used to specify the syntax of object-level expressions and give meaning to inference rules, is developed within ZF set theory. As in higher-order logic, it includes a simply typed λ-calculus over a given set of *ground types* G, with terms identified up to $\alpha\beta\eta$-conversion. To avoid confusion with the object-level, types of the metalanguage are called *meta-types*. In addition to the basic forms, constants may be defined so that the syntax of meta-terms is

$$e ::= x \mid c \mid \lambda x.e \mid e(e')$$

where x is a *meta-variable*, c is a constant, $\lambda x\,.\,e$ denotes *meta-abstraction* and $e(e')$ *meta-application*. Multiple abstractions $\lambda x_1\,.\,\lambda x_2\,.\,\ldots e$ are written as $\lambda x_1\ x_2 \ldots\,.\,e$, and multiple applications $e(e_1)(e_2)\ldots$ as $e(e_1, e_2, \ldots)$. Meta-types are constructed over the set of ground types G using $\Rightarrow$ for *meta-level function space*.

$$s ::= s_g \mid s \Rightarrow s' \qquad\qquad \textit{where } s_g \in G$$

Meta-type inhabitation is just set membership, written $e \in s$. There are the usual typing rules for abstraction and application, so that meta-level abstraction provides a uniform notation for variable binding in the syntax of object languages. Meta-types correspond to meta-types in the Isabelle implementation (see §4.1). The explicit use of set theory to give a semantics to CCL (e.g. Definition 3.2) is not required in the Isabelle implementation, which is just an axiomatisation of CCL.

Rules of an object theory are presented in a natural deduction style, including the use of hypothetical hypotheses [61]. As illustration, the following rules are taken from the first-order logic of §3.6.

$$\frac{P \qquad Q}{P \wedge Q}$$

states that for all P and Q, $P \wedge Q$ holds if the premises P and Q both hold. The following rule is hypothetical.

$$\frac{\exists\, x.\ P(x) \qquad \begin{matrix}\left[\ P(x)\ \right]_x \\ Q\end{matrix}}{Q}$$

It states that if $\exists\, x.\ P(x)$ holds, and $P(x)$ implies Q for all x, then Q holds. The brackets $[\ldots]_x$ indicate that the eigenvariable x is bound in the inference $P(x)$ *implies* Q, and cannot occur free in any other assumptions. When no hypothesis is present, eigenvariables annotate an overscore, for example

$$\frac{\overline{P(x)}^{\,x}}{\forall\, x.\ P(x)}$$

The following rule has a hypothetical hypothesis.

$$\frac{P \leftrightarrow Q \qquad \begin{matrix}\left[\begin{matrix}[\,Q\,] & & [\,P\,] \\ P & ; & Q\end{matrix}\right] \\ R\end{matrix}}{R}$$

It states that R holds if $P \leftrightarrow Q$ holds and if Q implies P and P implies Q then R holds. Hypothetical hypotheses can be annotated with eigenvariable conditions in the same way as hypotheses.

Logical equivalence at the meta-level ($\equiv$) is used for definitions. For example, the definition of $\leftrightarrow$ in first-order logic is

$$P \leftrightarrow Q \quad \equiv \quad (P \supset Q) \wedge (Q \supset P)$$

Before the logical connectives of CCL are introduced in §3.6, the symbols $\forall$, $\exists$, $\wedge$, $\vee$, $\neg$, $\Longrightarrow$ and $\leftrightarrow$ are used for the meta-logical connectives.

3.2 An Untyped Functional Language

This section describes the syntax and operational semantics of the untyped functional programming language $\mathcal{L}$. The language $\mathcal{L}$ can be regarded as an untyped λ-calculus sugared with constants for finite enumerations, pairs, general recursion and local declarations. The additional syntax allows an evaluation strategy to be defined for $\mathcal{L}$, and the subsequent imposition of a typing regime (see §3.4). Terms of the programming language $\mathcal{L}$ are represented by their abstract syntax trees; the meta-type ι is the set of all such terms. Constants of $\mathcal{L}$ are meta-functions over terms, and so the well-formedness of terms is ensured by the meta-type system.

First, the syntax of $\mathcal{L}$ is introduced, together with an informal description of the possible reductions. Then, an operational semantics is presented that embodies one particular evaluation strategy. *Determinacy* and *termination* are defined with respect to this semantics.

3.2.1 Syntax

Following the traditional functional programming parlance of McCarthy [41] and Landin [32], the term-formers of $\mathcal{L}$ are introduced in two groups: *constructors*, which build terms of particular structure; and *destructors*, which break up terms built by particular constructors.

Finite enumerations are constructed from the base term $\mathsf{z} \in \iota$ and the successor term-former $\mathsf{s} \in \iota \Rightarrow \iota$, for example z, $\mathsf{s}(\mathsf{z})$ and $\mathsf{s}(\mathsf{s}(\mathsf{z}))$. Functions are constructed using the term-former $\mathsf{lam} \in (\iota \Rightarrow \iota) \Rightarrow \iota$, and pairs using $\langle\rangle \in \iota \Rightarrow \iota \Rightarrow \iota$. Write $\mathsf{lam}(\lambda x . b(x))$ as $\mathsf{lam}\ x.b(x)$, and $\langle\rangle(a, b)$ as $\langle a, b\rangle$.

Each destructor reduces particular terms. As an informal motivation, these reductions are described using the relation $\rightsquigarrow$. The term $\mathsf{vcase} \in \iota$ does not reduce any term; it corresponds to an abort.[1] The term-former $\mathsf{scase} \in \iota \Rightarrow \iota \Rightarrow (\iota \Rightarrow \iota) \Rightarrow \iota$ reduces terms formed by z or s.

$$\mathsf{scase}(\mathsf{z}, b, c) \rightsquigarrow b$$
$$\mathsf{scase}(\mathsf{s}(a), b, c) \rightsquigarrow c(a)$$

The term-former $\mathsf{pcase} \in \iota \Rightarrow ((\iota \Rightarrow \iota) \Rightarrow \iota) \Rightarrow \iota$ reduces terms formed by lam.

$$\mathsf{pcase}(\mathsf{lam}\ x.b(x), c) \rightsquigarrow c(b)$$

[1]The destructors are named according to their corresponding type-formers, introduced in §3.4.

Reduction of lam, using pcase, follows the pattern of reduction for the other constructors; it is more general than that for function application (cf. the destructor $\mathsf{funsplit}$ in Martin-Löf's Type Theory [49]). In particular, pcase allows an induction rule (η-rule) to be derived (see §4.2.1). The term-former $\mathsf{split} \in \iota \Rightarrow (\iota \Rightarrow \iota \Rightarrow \iota) \Rightarrow \iota$ reduces terms formed by $\langle\rangle$.

$$\mathsf{split}(\langle a, b\rangle, c) \rightsquigarrow c(a, b)$$

These four destructors behave similarly to pattern matching in ML; compare the term $\mathsf{split}(p, \lambda x\ y.c(x, y))$ with the ML code

```
case p of <x,y> => c(x,y);
```

But the destructors of $\mathcal{L}$ are cruder than pattern matching in ML; they each match terms of only one data type, and every case must be considered. In addition to destructors for case analysis, two further destructors are introduced for recursion and local declarations. The term-former $\mathsf{rec} \in \iota \Rightarrow (\iota \Rightarrow (\iota \Rightarrow \iota) \Rightarrow \iota) \Rightarrow \iota$ reduces any term.

$$\mathsf{rec}(a, h) \rightsquigarrow h(a, \lambda x.\mathsf{rec}(x, h))$$

It allows general recursion to be expressed; the term $\mathsf{rec}(a, \lambda x\ g.h(x, g))$ corresponds to the ML code

```
let fun g(x) = h(x,g) in g(a) end;
```

Finally, the term-former $\mathsf{let} \in \iota \Rightarrow (\iota \Rightarrow \iota) \Rightarrow \iota$ introduces local definitions and forces the immediate evaluation of a term. Write $\mathsf{let}(a, \lambda x.b(x))$ as $\mathsf{let}\ x\ \mathsf{be}\ a\ \mathsf{in}\ b(x)\ \mathsf{end}$.

$$\frac{a \rightsquigarrow a'}{\mathsf{let}\ x\ \mathsf{be}\ a\ \mathsf{in}\ b(x)\ \mathsf{end} \rightsquigarrow b(a')}$$

Write *CON* for the set of constructors $\{\mathsf{z}, \mathsf{s}, \mathsf{lam}, \langle\rangle\}$, and *DES* for the set of destructors $\{\mathsf{vcase}, \mathsf{scase}, \mathsf{pcase}, \mathsf{split}, \mathsf{rec}, \mathsf{let}\}$.

3.2.2 Evaluation

The transition semantics informally described by $\rightsquigarrow$ gives an intuition for the destructors. Now, an Evaluation Semantics (Natural Semantics) is formally presented for $\mathcal{L}$. *Programs* are closed terms[2] of $\mathcal{L}$. *Evaluation* is defined as a particular strategy for reducing programs to *canonical* form.

Definition 3.1 *A closed term of $\mathcal{L}$ is canonical iff it has one of the following forms*

$$\mathsf{z} \qquad \mathsf{s}(a) \qquad \mathsf{lam}\ x.b(x) \qquad \langle a_1, a_2\rangle$$

for any terms a and meta-functions b.

Definition 3.2 *The evaluation relation $\rhd \in \iota \Rightarrow \iota \Rightarrow o$ is defined as the least relation satisfying the rules of Figure 3.1.*

[2]As usual, closed terms are those in which no variable occurs free.

$$\mathsf{z} \;\rhd\; \mathsf{z} \qquad\qquad \mathsf{s}(a) \;\rhd\; \mathsf{s}(a)$$

$$\mathsf{lam}\ x.b(x) \;\rhd\; \mathsf{lam}\ x.b(x) \qquad\qquad \langle a, b\rangle \;\rhd\; \langle a, b\rangle$$

$$\frac{u \;\rhd\; \mathsf{z} \qquad v \;\rhd\; a}{\mathsf{scase}(u, v, w) \;\rhd\; a} \qquad\qquad \frac{u \;\rhd\; \mathsf{s}(a) \qquad w(a) \;\rhd\; b}{\mathsf{scase}(u, v, w) \;\rhd\; b}$$

$$\frac{u \;\rhd\; \mathsf{lam}\ x.b(x) \qquad v(b) \;\rhd\; a}{\mathsf{pcase}(u, v) \;\rhd\; a} \qquad\qquad \frac{u \;\rhd\; \langle a, b\rangle \qquad v(a, b) \;\rhd\; c}{\mathsf{split}(u, v) \;\rhd\; c}$$

$$\frac{v(u, \lambda x.\mathsf{rec}(x, v)) \;\rhd\; a}{\mathsf{rec}(u, v) \;\rhd\; a} \qquad\qquad \frac{u \;\rhd\; a \qquad v(a) \;\rhd\; b}{\mathsf{let}\ x\ \mathsf{be}\ u\ \mathsf{in}\ v(x)\ \mathsf{end} \;\rhd\; b}$$

Figure 3.1: *Evaluation Rules for $\mathcal{L}$*

In $\mathcal{L}$, function application is a derived form. Call-by-name ‘ $\in \iota \Rightarrow \iota \Rightarrow \iota$ and call-by-value application ^ $\in \iota \Rightarrow \iota \Rightarrow \iota$ are defined by

$$\begin{aligned} f \,‘\, a &\equiv \mathsf{pcase}(f, \lambda x.x(a)) \\ f \,\hat{}\, a &\equiv \mathsf{let}\ x\ \mathsf{be}\ a\ \mathsf{in}\ f \,‘\, x\ \mathsf{end} \end{aligned}$$

For call-by-value, let forces the immediate evaluation of a. From these definitions, the following evaluation rules are derived.

$$\frac{u \;\rhd\; \mathsf{lam}\ x.b(x) \qquad b(v) \;\rhd\; a}{u \,‘\, v \;\rhd\; a}$$

$$\frac{u \;\rhd\; \mathsf{lam}\ x.b(x) \qquad v \;\rhd\; a \qquad b(a) \;\rhd\; c}{u \,\hat{}\, v \;\rhd\; c}$$

In keeping with a lazy regime, only the call-by-name form (‘) is used.

The definition of $\rhd$ provides a Prolog-style interpreter for $\mathcal{L}$; evaluating a program t to canonical form a corresponds to proving the theorem $t \;\rhd\; a$ by depth-first search over the evaluation rules. If a is initially uninstantiated, then it will become instantiated during the proof.

For a fixed evaluation strategy, there is no need of theorems for confluence (Church-Rosser) and strong normalisation. Instead, *determinacy* and *termination* are considered. If a program can be evaluated to canonical form, then there is a unique result and a unique derivation.

Theorem 3.1 (Determinacy) *For any program t and canonical terms a and a', if $t \;\rhd\; a$ and $t \;\rhd\; a'$ then a and a' (and their derivations) are identical.*

Proof: By rule induction over the evaluation rules of Figure 3.1. Operators for which there is only one rule obviously preserve determinacy. For scase, the induction hypothesis ensures that at most one of the two rules is applicable as the premises $u \rhd \mathsf{z}$ and $u \rhd \mathsf{s}(a)$ are mutually exclusive. ■

In $\mathcal{L}$, the evaluation of program t to canonical form a corresponds to the derivation $\vdash t \rhd a$. Attempting to evaluate a program that fails to terminate either reaches a non-canonical term that cannot be reduced (e.g. vcase), or leads to a derivation that also fails to terminate. Write $t\!\downarrow$ iff program t terminates.

Definition 3.3 (Termination) *For any program t, $t\!\downarrow$ iff $\exists\, a \in \iota.\ t \rhd a$*

Clearly, not all programs terminate. Consider the following terms.

$$\mathsf{vcase} \qquad\qquad \mathsf{split}(\mathsf{z}, \lambda x\ y.x) \qquad\qquad \mathsf{rec}(a, \lambda x\ g.g(x))$$

and the paradoxical combinator

$$\Omega \ \equiv\ \mathsf{lam}\ x.(x\ \text{`}\ x)\ \text{`}\ \mathsf{lam}\ x.(x\ \text{`}\ x)$$

3.3 Term Equality

To reason about programs conveniently, a type-less equality is required that reflects the operational semantics, capturing our intuitive notion of when programs are the same. For a lazy regime, *applicative bisimulation* is an appropriate relation [1].

A pre-order $\preccurlyeq$ is defined over terms, and *term equality* $=$ is defined as the free equivalence generated by $\preccurlyeq$. Using results from Howe [28], the equivalence relation is shown to be a congruence.

3.3.1 A Pre-Order Over Terms

The *arity* of a meta-function f, written $\alpha(f)$, is the sequence of meta-types of its arguments (e.g. $\alpha(\mathsf{split})$ is $[\iota, \iota \Rightarrow \iota \Rightarrow \iota]$). Write $\overline{x}$ for a possibly empty sequence of arguments. For a relation R over terms, write $\overline{x} R \overline{x}'$ for sequences of equal length, when the relation R holds pointwise over the arguments in each sequence; and write $f R f'$ for meta-functions f and f' of the same arity, when the relation R holds for all applications (i.e. $\forall\, \overline{x} \in \alpha(f).\ f(\overline{x}) R f'(\overline{x})$).

Term equality is defined using the pre-order $\preccurlyeq\ \in \iota \Rightarrow \iota \Rightarrow o$. Informally, $t \preccurlyeq t'$ iff t fails to terminate or t and t' evaluate to terms with the same outermost constructor and components $\overline{x}$ and $\overline{x}'$ for which $\overline{x} \preccurlyeq \overline{x}'$. Formally, the relation is defined as the greatest fixed point of the function $[\cdot]$, defined over relations as follows.

Definition 3.4 $[\cdot]$ *is a mapping over term relations such that for all relations R defined over closed terms and for all terms t and t'*

$$t\,[R]\,t' \quad \textit{iff} \quad \forall\, c \in CON.\ \forall\, \overline{x} \in \alpha(c).\ t \rhd c(\overline{x}) \Longrightarrow (\exists\, \overline{x}' \in \alpha(c).\ t' \rhd c(\overline{x}') \ \wedge\ \overline{x}\, R\, \overline{x}')$$

The function $[\cdot]$ is monotonic (i.e. $\forall\, R\ R'.\ R \subseteq R' \ \supset\ [R] \subseteq [R']$), so that by the Knaster-Tarski Theorem [15] there is a greatest fixed point given by the following definition.

Definition 3.5 $\preccurlyeq\ \equiv\ \bigcup\{\ R$ *over closed terms* $\mid R \subseteq [R]\ \}$

Therefore, for all closed terms t and t'

$$t \preccurlyeq t' \quad \mathit{iff} \quad \forall c \in \mathit{CON}.\, \forall \overline{x} \in \alpha(c).\ t \ \rhd\ c(\overline{x}) \Longrightarrow (\exists \overline{x}' \in \alpha(c).\ t' \ \rhd\ c(\overline{x}') \ \wedge\ \overline{x} \preccurlyeq \overline{x}')$$

Theorem 3.2 $\preccurlyeq$ *is a pre-order.*

Proof:

- *Reflexivity* ($t \preccurlyeq t$): The identity relation (syntactic equality $\equiv$) is contained in $\preccurlyeq$, since $\equiv\ \subseteq\ [\equiv]$.
- *Transitivity* ($t \preccurlyeq u\ \wedge\ u \preccurlyeq v\ \Longrightarrow\ t \preccurlyeq v$): Let $\circ$ represent relational composition. To prove that $\preccurlyeq$ is transitive, it suffices to prove $[R] \circ [S]\ \subseteq\ [R \circ S]$ for all relations R and S, since this implies $(\preccurlyeq \circ \preccurlyeq)\ \subseteq\ \preccurlyeq$.
  So, prove $t\,[R \circ S]\,v$ assuming $t\,([R] \circ [S])\,v$, that is there exists u such that

  (1) $$t\,[R]\,u \qquad \mathit{and} \qquad u\,[S]\,v$$

  By the definition of $[\cdot]$, if t fails to terminate then $t\,[R \circ S]\,v$ trivially holds, otherwise (1) implies that u terminates which in turn implies that v terminates and for some $c \in \mathit{CON}$

  (2) $$t\ \rhd\ c(\overline{x}) \qquad \mathit{and} \qquad u\ \rhd\ c(\overline{x}') \qquad \mathit{and} \qquad \overline{x}\,R\,\overline{x}'$$

  (3) $$u\ \rhd\ c(\overline{x}') \qquad \mathit{and} \qquad v\ \rhd\ c(\overline{x}'') \qquad \mathit{and} \qquad \overline{x}'\,S\,\overline{x}''$$

  The term $c(\overline{x}')$ is common to both since evaluation is unique. Therefore,

  (4) $$t\ \rhd\ c(\overline{x}) \qquad \mathit{and} \qquad v\ \rhd\ c(\overline{x}'') \qquad \mathit{and} \qquad \overline{x}\,(R \circ S)\,\overline{x}''$$

  and so $t\,[R \circ S]\,v$. ∎

Although $\preccurlyeq$ is defined by propagating evaluation through the constructors, this is done in a pointwise fashion so that $\preccurlyeq$ is stronger than the corresponding definition for an eager evaluation scheme $\preccurlyeq_e$. For example, consider

$$\langle \Omega, \mathsf{z} \rangle \not\preccurlyeq \langle \Omega, \mathsf{s}(\mathsf{z}) \rangle \qquad \mathit{but} \qquad \langle \Omega, \mathsf{z} \rangle \preccurlyeq_e \langle \Omega, \mathsf{s}(\mathsf{z}) \rangle$$

Moreover, $t \preccurlyeq t'$ does not imply that the terms t and t' have the same canonical form. Consider the following distinct canonical terms, which satisfy the relation.

$$\langle \mathsf{lam}\ x.x\ \text{`}\ \mathsf{z}, \mathsf{z} \rangle \preccurlyeq \langle \mathsf{z}, \mathsf{z} \rangle$$

However, if $t \preccurlyeq t'$ holds for a program t that terminates, then t' also terminates and the two canonical forms have the same outermost constructor.

Theorem 3.3 *For all closed terms t and t', if $t \preccurlyeq t'$ then $t\!\downarrow$ implies $t'\!\downarrow$.*

Proof: As t terminates, the definition of $\preccurlyeq$ implies that there exists a term a' such that $t'\ \rhd\ a'$, that is t' terminates. ∎

The relation $\preccurlyeq$ is shown to be a congruence. Howe [28] proved that if an extensionality condition holds for each of the operators of a lazy computation system, then the equivalence relation is a congruence. This result is directly applicable to all the operators of $\mathcal{L}$ except rec. The proofs of extensionality closely follow the examples presented by Howe and so are not given here.

Theorem 3.4 *For all closed meta-functions f and closed sequences $\overline{x}, \overline{x}'$ such that $f(\overline{x})$ and $f(\overline{x}')$ are well-formed, closed terms*

$$\overline{x} \preccurlyeq \overline{x}' \quad \Longrightarrow \quad f(\overline{x}) \preccurlyeq f(\overline{x}')$$

Proof: For the operators

$$\mathsf{z},\ \mathsf{s},\ \mathsf{lam},\ \langle\rangle,\ \mathsf{scase},\ \mathsf{pcase},\ \mathsf{split},\ \mathsf{let}$$

the proof of congruence follows the details of Howe [28]. For each, the extensionality condition is proved (in the same way as the examples presented by Howe) and congruence then inferred.

The evaluation rule for rec is not suited to this approach. Instead, congruence for rec is proved directly, that is

$$a \preccurlyeq a' \ \wedge \ h \preccurlyeq h' \ \Longrightarrow \ \mathsf{rec}(a,h) \preccurlyeq \mathsf{rec}(a',h')$$

If $\mathsf{rec}(a,h)$ fails to terminate this holds immediately. Otherwise

$$\exists\, d.\ \mathsf{rec}(a,h) \ \rhd \ d \qquad \textit{in } k \textit{ steps}$$

By induction on the length of the evaluation, assume

$$\mathsf{rec}(x,h) \preccurlyeq \mathsf{rec}(x,h')$$

holds for all $\mathsf{rec}(x,h)$ that evaluate in $< k$ steps. From the evaluation rule for rec, $\mathsf{rec}(a,h) \preccurlyeq \mathsf{rec}(a',h')$ holds if

$$(*) \qquad\qquad h(a, \lambda x.\mathsf{rec}(x,h)) \preccurlyeq h'(a', \lambda x.\mathsf{rec}(x,h'))$$

Either $\mathsf{rec}(x,h)$ is not evaluated in the execution of $h(a,\lambda x.\mathsf{rec}(x,h))$ in which case its value is irrelevant and $(*)$ holds, or $\mathsf{rec}(x,h) \ \rhd \ e$ in under k steps, $\mathsf{rec}(x,h) \preccurlyeq \mathsf{rec}(x,h')$ follows from the induction hypothesis and again $(*)$ holds.

■

3.3.2 An Equivalence Over Terms

The infix relation $= \ \in \iota \Rightarrow \iota \Rightarrow o$ is defined as the free equivalence generated by the pre-order $\preccurlyeq$.

Definition 3.6 *For all closed terms t and t',* $\quad t = t' \quad$ *iff* $\quad t \preccurlyeq t' \ \wedge \ t' \preccurlyeq t.$

Theorem 3.5 (Equivalence) $=$ *is an equivalence relation.*

Proof: As $\preccurlyeq$ is a pre-order, it immediately follows that $=$ is reflexive, transitive and symmetric. ■

The meta-type $\iota_=$ is defined as the set of equivalence classes of terms (ι) under this relation.

Definition 3.7 $\iota_=$ *is the set* $\quad \iota/=$

As with $\preccurlyeq$, the relation $=$ does not respect canonical forms. The termination and congruence results for $\preccurlyeq$ are symmetric and so carry over to $=$.

Theorem 3.6 (Termination) *For all closed terms t and t', if $t = t'$, then $t\!\downarrow$ iff $t'\!\downarrow$.*

Theorem 3.7 (Congruence) *For all closed meta-functions f and closed sequences $\overline{x}, \overline{x}'$ such that $f(\overline{x})$ and $f(\overline{x}')$ are well-formed, closed terms*

$$\overline{x} = \overline{x}' \quad \Longrightarrow \quad f(\overline{x}) = f(\overline{x}')$$

The definition of $\preccurlyeq$, and therefore that of $=$, reflect the evaluation of terms. Conversion rules can be derived that correspond to transitions in the operational semantics. For the destructor let, a conversion rule is derived only for terminating arguments.

Theorem 3.8 (Conversion) *For all terms and meta-functions a, b, c, h the following conversions hold for well-formed, closed terms.*

- $\mathsf{scase}(\mathsf{z}, b, c) = b$
- $\mathsf{scase}(\mathsf{s}(a), b, c) = c(a)$
- $\mathsf{pcase}(\mathsf{lam}\ x.b(x), c) = c(b)$
- $\mathsf{split}(\langle a, b\rangle, c) = c(a, b)$
- $\mathsf{rec}(a, h) = h(a, \lambda x.\mathsf{rec}(x, h))$
- $a\!\downarrow \implies \mathsf{let}\ x\ \mathsf{be}\ a\ \mathsf{in}\ b(x)\ \mathsf{end} = b(a)$

Proof: By the definition of equality, the formula $a = b$ holds iff $a \preccurlyeq b$ and $b \preccurlyeq a$. In each case, if the left-hand side fails to terminate, then the relation trivially holds; otherwise there exists a unique derivation. Except for the rule for let, evaluation is lazy and the conversions are simple to prove. Three of the twelve cases are illustrated.

- $\mathsf{pcase}(\mathsf{lam}\ x.b(x), c) \preccurlyeq c(b)$

  There exists a derivation for the evaluation of the left-hand side that concludes with

$$\frac{\mathsf{lam}\ x.b(x) \ \rhd\ \mathsf{lam}\ x.b(x) \qquad c(b)\ \rhd\ a}{\mathsf{pcase}(\mathsf{lam}\ x.b(x), c)\ \rhd\ a}$$

  for some a. Therefore,

$$\mathsf{pcase}(\mathsf{lam}\ x.b(x), c)\ \rhd\ a \qquad \textit{and} \qquad c(b)\ \rhd\ a \qquad \textit{and} \qquad a \preccurlyeq a$$

- $c(b) \preccurlyeq \mathsf{pcase}(\mathsf{lam}\ x.b(x), c)$

  There exists a unique canonical form a for the left-hand side

$$c(b)\ \rhd\ a$$

  By the evaluation rule for pcase, $c(b)\ \rhd\ a$ and $\mathsf{lam}\ x.b(x)\ \rhd\ \mathsf{lam}\ x.b(x)$ imply

$$\mathsf{pcase}(\mathsf{lam}\ x.b(x), c)\ \rhd\ a$$

- $a\downarrow \implies b(a) \preccurlyeq \mathsf{let}\ x\ \mathsf{be}\ a\ \mathsf{in}\ b(x)\ \mathsf{end}$

  There exists a unique canonical form c for the left-hand side

  $$b(a) \ \rhd\ c$$

  From the premise $a\downarrow$, there exists a term a' such that

  $$a \ \rhd\ a'$$

  This implies $a \preccurlyeq a'$ which in turn implies $b(a) \preccurlyeq b(a')$ by the congruence result for $\preccurlyeq$ (Theorem 3.4). Therefore, there exists a term c' such that

  $$b(a') \ \rhd\ c' \qquad \textit{and} \qquad c \preccurlyeq c'$$

  By the evaluation rule for let, this implies

  $$\mathsf{let}\ x\ \mathsf{be}\ a\ \mathsf{in}\ b(x)\ \mathsf{end} \ \rhd\ c'$$

■

Theorems 3.6, 3.7 and 3.8 are proved for term equality defined over closed terms. Now the definition of equality is extended to open terms in a uniform way.

Definition 3.8 *For all open terms t and t', $t = t'$ iff for all substitutions σ such that $\sigma(t)$ and $\sigma(t')$ are closed $\sigma(t) \preccurlyeq \sigma(t')$.*

It is straightforward to carry through the results for termination, congruence and conversion to open terms.

3.4 Types for $\mathcal{L}$

As term equality ($=$) is a congruence relation over the term-formers of ι (Theorem 3.7), the elements of $\iota_=$ can be considered as values, and the term-formers of ι lifted up to term-formers of $\iota_=$. The symbols z, s, lam, $\langle\rangle$, vcase, scase, pcase, split, rec and let are now used as the corresponding term-formers for $\iota_=$.

Types are sets of equivalence classes of terminating programs. They formalise the observation that terminating programs evaluate to finite enumerations, functions and pairs; and allow inductive reasoning over the structure of programs. For products and functions, generalised (dependent) type-formers are used, as these allow a fine grain treatment of termination, and are convenient for specifying input/output behaviour (see §1.3). Inductive types are defined as least fixed points of monotonic functions; the predicate *Mono* formalises monotonicity for meta-functions $\tau \Rightarrow \tau$. Two destructors (scase and split) are defined for types in such a way that they behave as their namesakes for terms.

First, the syntax of types is presented with informal descriptions. Then, the type-formers are defined. Finally, predicates are defined for type inhabitation ($a\!:\!A$), the monotonicity of meta-functions ($\mathit{Mono}(\lambda X\,.\,B(X))$) and type equality ($A = B$)—overloading the symbol $=$, which is already used for equality between terms.

3.4.1 Syntax

Finite enumeration types are constructed from the empty (void) type $\mathsf{V} \in \tau$ and the successor type-former $\mathsf{S} \in \tau \Rightarrow \tau$, for example $\mathsf{V}, \mathsf{S}(\mathsf{V})$ and $\mathsf{S}(\mathsf{S}(\mathsf{V}))$ are the types of zero, one and two elements respectively. Generalised functions are constructed using the type-former $\Pi \in \tau \Rightarrow (\iota_= \Rightarrow \tau) \Rightarrow \tau$, and generalised products using $\Sigma \in \tau \Rightarrow (\iota_= \Rightarrow \tau) \Rightarrow \tau$. Write $\Pi(A, B)$ and $\Sigma(A, B)$ as $\Pi x{:}A.B(x)$ and $\Sigma x{:}A.B(x)$, and when there is no dependency as $A \rightarrow B$ and $A \times B$ respectively. Elements of a general function type are functions, for which the type of the codomain depends on the value to which the function is applied; elements of a general product type are pairs, for which the type of the second element depends on the value of the first. Inductive types are constructed as least fixed points of meta-functions over types, using the type-former $\mu \in (\tau \Rightarrow \tau) \Rightarrow \tau$ and the predicate $\mathit{Mono} \in (\tau \Rightarrow \tau) \Rightarrow o$. Write $\mu(\lambda X\,.\,B(X))$ as $\mu\ X\,.\,B(X)$. Finally, arbitrary subtypes are constructed using the type-former $\{\} \in \tau \Rightarrow (\iota_= \Rightarrow o) \Rightarrow \tau$. Write $\{\}(A, \lambda x\,.\,P(x))$ as $\{x{:}A, P(x)\}$.

Two destructors are introduced that reduce to types rather than terms (note the overloading with the destructors for terms).

$$\begin{aligned} \mathsf{scase} &\in \iota_= \Rightarrow \tau \Rightarrow (\iota_= \Rightarrow \tau) \Rightarrow \tau \\ \mathsf{split} &\in \iota_= \Rightarrow (\iota_= \Rightarrow \iota_= \Rightarrow \tau) \Rightarrow \tau \end{aligned}$$

3.4.2 Types and Type-Formers

Types are defined as sets of equivalence classes of terminating terms. The meta-type τ is the set of all types.

Definition 3.9 $\tau \;\equiv\; \{\, A \in \mathcal{P}(\iota_=) \mid \forall x{\in}A.\ x{\downarrow} \,\}$

where $\mathcal{P}(A)$ is the power set of A. The definition make sense because term equality respects termination (Theorem 3.3), and so the predicate $\downarrow$ can be lifted up to equivalences classes of terms ($\iota_=$). The following two theorems are immediate consequences of this definition.

Theorem 3.9 *For all A and B such that $B \subseteq A$, $A \in \tau$ implies $B \in \tau$.*

Theorem 3.10 *For all terms a and types A, $a \in A$ implies $a{\downarrow}$.*

Definition 3.10 *The type-formers are defined as follows.*

$$\begin{aligned} \mathsf{V} &\equiv \emptyset \\ \mathsf{S}(A) &\equiv \{t \in \iota_= \mid\ t = \mathsf{z}\ \vee\ (\exists\, a.\ t = \mathsf{s}(a) \wedge a \in A)\ \} \\ \Pi(A, B) &\equiv \{t \in \iota_= \mid\ \exists\, b.\ t = \mathsf{lam}\ x.b(x)\ \wedge\ \forall x.\ x \in A \Longrightarrow b(x) \in B(x)\ \} \\ \Sigma(A, B) &\equiv \{t \in \iota_= \mid\ \exists\, a\ b.\ t = \langle a, b\rangle\ \wedge\ a \in A \wedge b \in B(a)\ \} \\ \mu\, X\,.\,B(X) &\equiv \bigcap\{X \in \tau\ \mid\ B(X) \subseteq X\ \} \\ \{x{:}A, P(x)\} &\equiv \{x \in A \mid\ P(x)\ \} \end{aligned}$$

where set intersection is the meet operation on the complete poset $(\tau, \subseteq)$. Note that the quantifiers in the above definitions are part of the meta-logic. In particular, the variable

b in the definition of $\mathsf{\Pi}$ is of meta-type $\iota_= \Rightarrow \iota_=$. From these definitions, the meta-types for the type-formers described above are proved.

Theorem 3.11 *The type-formers have the following meta-types*

- $\mathsf{V} \in \tau$
- $\mathsf{S} \in \tau \Rightarrow \tau$
- $\mathsf{\Pi} \in \tau \Rightarrow (\iota_= \Rightarrow \tau) \Rightarrow \tau$
- $\mathsf{\Sigma} \in \tau \Rightarrow (\iota_= \Rightarrow \tau) \Rightarrow \tau$
- $\mu \in (\tau \Rightarrow \tau) \Rightarrow \tau$
- $\{\} \in \tau \Rightarrow (\iota_= \Rightarrow o) \Rightarrow \tau$

Proof: The empty set V is clearly a member of τ.
For the type-formers S, $\mathsf{\Pi}$ and $\mathsf{\Sigma}$, their definitions imply that all members are equal to a canonical term, and therefore terminate (Theorem 3.6).
For the type-former $\{\}$, every subset of a type is a type (Theorem 3.9), so $A \in \tau$ implies $\{x{:}A, P(x)\} \in \tau$.
For the type-former μ, the set of types (τ) is closed under arbitrary subsets, and therefore under intersection so that $\mu(B) \in \tau$. ■

3.4.3 Predicates for Types

Predicates are defined for monotonicity, type equality and type inhabitation. From the definitions, rules are derived for reasoning about the predicates.

The predicate $Mono(f)$ holds iff the meta-function $f \in \tau \Rightarrow \tau$ is monotonic.

Definition 3.11 $Mono(f)$ *iff* $\forall x\ y {\in} \tau.\ x \subseteq y \Longrightarrow f(x) \subseteq f(y)$

Type equality ($= \in \tau \Rightarrow \tau \Rightarrow o$) is just set equality. The destructors for types are defined to satisfy the same conversion rules as their namesakes for terms.

Definition 3.12

$$\begin{aligned}\mathsf{scase}(a, B, C) &\equiv \{x \in \tau \mid \ (a = \mathsf{z}\ \wedge\ x = B)\ \vee\ (\exists\, y.\ a = \mathsf{s}(y) \wedge x = C(y))\ \}\\ \mathsf{split}(p, C) &\equiv \{x \in \tau \mid \ \exists\, a\ b.\ p = \langle a, b\rangle\ \wedge\ x = C(a, b)\ \}\end{aligned}$$

Type inhabitation is set membership.

Definition 3.13 $a : A$ *iff* $a \in A$

$$Mono(\lambda X.X) \qquad\qquad Mono(\lambda X.A)$$

$$\frac{Mono(\lambda X.A(X))}{Mono(\lambda X.\mathsf{S}(A(X)))} \qquad\qquad \frac{\begin{matrix}\left[\; x : A \;\right]_x \\ Mono(\lambda X.B(X,x))\end{matrix}}{Mono(\lambda X.\mathsf{\Pi}(A, B(X)))}$$

$$\frac{Mono(\lambda X.A(X)) \qquad \begin{matrix}\left[\; x : A(X) \;\right]_{x,X} \\ Mono(\lambda X.B(X,x))\end{matrix}}{Mono(\lambda X.\mathsf{\Sigma}(A(X), B(X)))} \qquad \frac{Mono(\lambda X.A(X))}{Mono(\lambda X.\{x{:}A(X), P(x)\})}$$

Figure 3.2: *Monotonicity Rules for CCL*

Theorem 3.12 (Monotonicity) *The monotonicity rules of Figure 3.2 are derivable.*

Proof: From the definition of *Mono*, it immediately follows that the identity function $\lambda X.X$ and the constant function $\lambda X.A$ are monotonic. The other rules follow from the definitions of the type-formers. For example,

- Prove that $X \subseteq Y \implies \mathsf{\Pi}(A, B(X)) \subseteq \mathsf{\Pi}(A, B(Y))$ for all $X, Y \in \tau$, assuming

  (1) $$\forall x{\in}A.\ \forall X\ Y{\in}\tau.\ X \subseteq Y \implies B(X,x) \subseteq B(Y,x)$$

  It suffices to prove that for an arbitrary element $a : \mathsf{\Pi}(A, B(X))$ it follows that $a : \mathsf{\Pi}(A, B(Y))$.
  From the definition of $\mathsf{\Pi}$, there exists a meta-function b such that

  (2) $$a = \mathsf{lam}\ x.b(x) \ \wedge\ \forall x.\ x \in A \implies b(x) \in B(X,x)$$

  Since $X \subseteq Y$, (1) implies that $B(X,x) \subseteq B(Y,x)$ and, therefore, that

  (3) $$\forall x.\ x \in A \implies b(x) \in B(Y,x)$$

  Hence, $a \in \mathsf{\Pi}(A, B(Y))$. ∎

The conversion rules for scase and split immediately follow from Definition 3.12. The predicate *Mono* justifies the expansion of inductive types. By the Knaster-Tarski Theorem [15], if a function f is monotonic (i.e. $Mono(f)$ holds), then it has a least fixed-point $lfp(f)$ such that $lfp(f) = f(lfp(f))$.

Theorem 3.13 (Type Conversion) *The following conversions hold between types.*

- $\mathsf{scase}(\mathsf{z}, B, C) = B$
- $\mathsf{scase}(\mathsf{s}(a), B, C) = C(a)$
- $\mathsf{split}(\langle a, b\rangle, C) = C(a, b)$
- $Mono(\lambda X.B(X)) \implies \mu X.\ B(X) = B(\mu X.\ B(X))$

From the definitions of the type-formers, a set of type rules is derived for the term-formers of $\mathcal{L}$ (z-*type*, s-*type*, etc.).

Theorem 3.14 *The type rules of Figure 3.3 are derivable.*

Proof: Proofs of three of the rules serve as illustration. For rec-*type*, the definition of the predicate $\prec_R$ is assumed; in §3.5, $a \prec_R b$ holds iff R is well-founded over $\iota_=$, and the pair (a, b) is in the ordering R.

- lam-*type*

  Prove $\mathsf{lam}\ x.b(x) \in \Pi(A, B)$, assuming

  (1) $$\forall x.\ x \in A \implies b(x) \in B(x)$$

  By the definition of Π, (1) and $\mathsf{lam}\ x.b(x) = \mathsf{lam}\ x.b(x)$ imply

  $$\mathsf{lam}\ x.b(x) \in \Pi(A, B)$$

- pcase-*type*

  Prove $\mathsf{pcase}(f, c) \in C(f)$, assuming

  (1) $$f \in \Pi(A, B)$$

  (2) $$\forall u.\ (\forall x.\ x \in A \implies u(x) \in B(x)) \implies c(u) \in C(\mathsf{lam}\ x.u(x))$$

  By the definition of Π, (1) implies there exists u' such that

  (3) $$f = \mathsf{lam}\ x.u'(x) \qquad \textit{and} \qquad \forall x.\ x \in A \implies u'(x) \in B(x)$$

  By (2), this in turn implies

  (4) $$c(u') \in C(\mathsf{lam}\ x.u'(x))$$

  By substitution, using (3) and $\mathsf{pcase}(\mathsf{lam}\ x.u'(x), c) = c(u')$ from Theorem 3.8, this gives

  $$\mathsf{pcase}(f, c) \in C(f)$$

- rec-*type*

  Prove $\mathsf{rec}(a, h) \in B(a)$, assuming

  (1) $$a \in A$$

  (2) $$\forall x\ u.\ x \in A \land (\forall y.\ y \in A \land y \prec_R x \implies u(y) \in B(y)) \implies h(x, u) \in B(x)$$

  Note that the well-formedness of the rule ensures that R is well-founded over $\iota_=$. So, by well-founded induction, it suffices to prove $\mathsf{rec}(a', h) \in B(a')$, assuming

  (3) $$a' \in A$$

  (4) $$\forall u.\ u \in A \land u \prec_R a' \implies \mathsf{rec}(u, h) \in B(u)$$

  By (2), (3) and (4) imply

  (5) $$h(a', \lambda y.\mathsf{rec}(y, h)) \in B(a')$$

  By substitution, using $\mathsf{rec}(a', h) = h(a', \lambda x.\mathsf{rec}(x, h))$ from Theorem 3.8, this gives

  $$\mathsf{rec}(a', h) \in B(a')$$

■

$$\frac{a : \mathsf{V}}{\mathsf{vcase} : A}$$

$$\mathsf{z} : \mathsf{S}(A) \qquad\qquad \frac{a : A}{\mathsf{s}(a) : \mathsf{S}(A)} \qquad\qquad \frac{a : \mathsf{S}(A) \qquad b : B(\mathsf{z}) \qquad \begin{matrix} \left[\; x : A \;\right]_x \\ c(x) : B(\mathsf{s}(x)) \end{matrix}}{\mathsf{scase}(a, b, c) : B(a)}$$

$$\frac{\begin{matrix} \left[\; x : A \;\right]_x \\ b(x) : B(x) \end{matrix}}{\mathsf{lam}\ x.b(x) : \Pi(A, B)} \qquad\qquad \frac{f : \Pi(A, B) \qquad \begin{matrix} \left[\; \begin{matrix} [\, x : A \,]_x \\ u(x) : B(x) \end{matrix} \;\right]_u \\ c(u) : C(\mathsf{lam}\ x.u(x)) \end{matrix}}{\mathsf{pcase}(f, c) : C(f)}$$

$$\frac{a : A \qquad b : B(a)}{\langle a, b \rangle : \Sigma(A, B)} \qquad\qquad \frac{p : \Sigma(A, B) \qquad \begin{matrix} \left[\; x : A\ ;\ \ y : B(x) \;\right]_{x,y} \\ c(x, y) : C(\langle x, y \rangle) \end{matrix}}{\mathsf{split}(p, c) : C(p)}$$

$$\frac{a : A \qquad \begin{matrix} \left[\; x : A\ ; \quad \begin{matrix} [\, y : A\ ;\ \ y \prec_R x \,]_y \\ u(y) : B(y) \end{matrix} \;\right]_{x,u} \\ h(x, u) : B(x) \end{matrix}}{\mathsf{rec}(a, h) : B(a)} \qquad\qquad \frac{a : A \qquad \begin{matrix} \left[\; x : A \;\right]_x \\ b(x) : B \end{matrix}}{\mathsf{let}\ x\ \mathsf{be}\ a\ \mathsf{in}\ b(x)\ \mathsf{end} : B}$$

$$\frac{a : A \qquad P(a)}{a : \{x{:}A, P(x)\}} \qquad\qquad \frac{a : \{x{:}A, P(x)\} \qquad \begin{matrix} \left[\; a : A\ ;\ \ P(a) \;\right] \\ Q \end{matrix}}{Q}$$

Figure 3.3: *Type Rules for CCL*

As types are defined over equivalence classes of terms under term equality, equal terms may be substituted. Moreover, type equality is just set equality, and therefore extensional. The following theorems hold for substitution.

Theorem 3.15 *For all terms a and a' and meta-functions C, if $a = a'$ then $C(a) = C(a')$.*

Theorem 3.16 *For all types A, A' and meta-functions C, if $A = A'$ then $C(A) = C(A')$.*

Theorem 3.17 *For all terms a and a' and types A and A', if $a = a'$ and $A = A'$ then $a : A$ iff $a' : A'$.*

3.5 Well-Founded Orderings

In §A.2, certain constructions over relations are shown to preserve well-foundedness. They are used as constructors for the meta-type of well-founded orderings (ω), so that well-foundedness is a syntactic condition, which depends only on the well-formedness of elements of ω.

The primitive ordering pR is the irreflexive transitive closure of the immediate subterm relation on terms. For example, the following hold.

$$x \prec_{\mathsf{pR}} \mathsf{s}(x) \qquad x \prec_{\mathsf{pR}} \mathsf{s}(\mathsf{s}(x)) \qquad x \prec_{\mathsf{pR}} \langle x, y\rangle$$

The ordering $\mathsf{lex}(R, S)$ is the lexicographic ordering of R and S; $\mathsf{map}(f, R)$ is the image of the ordering R under the meta-function f; and $\mathsf{restrict}(A, R)$ is the ordering R restricted to the type A.

Formally, the meta-type ω is the set of well-founded relations over terms.

Definition 3.14 $\omega \;\equiv\; \{R \in \mathcal{P}(\iota_= \times \iota_=) \mid \mathit{Wfd}_{\iota_=}(R)\}$

The predicate $\mathit{Wfd}_A(R)$, defined in §A.1, asserts that R is a well-founded relation over the type A.

Definition 3.15 *The constructions for well-founded orderings are defined as follows.*

$$\begin{array}{rcl}
\mathsf{pR} & \equiv & \bigcup_{a,b\in\iota_=} \{\ (a, \mathsf{s}(a)),\ (a, \langle a, b\rangle),\ (b, \langle a, b\rangle)\ \}^+ \\
\mathsf{lex}(R, S) & \equiv & \{\ (\langle a_1, a_2\rangle, \langle b_1, b_2\rangle) \in \iota_= \times \iota_= \mid\ (a_1, b_1) \in R \ \vee\ (a_1 = b_1 \wedge (a_2, b_2) \in S)\ \} \\
\mathsf{map}(f, R) & \equiv & \{\ (a, b) \in \iota_= \times \iota_= \mid\ (f(a), f(b)) \in R\ \} \\
\mathsf{restrict}(A, R) & \equiv & R\ \cap\ A \times A
\end{array}$$

Theorem 3.18 *The constructors for well-founded orderings have the following meta-types.*

- $\mathsf{pR} \ \in\ \omega$
- $\mathsf{lex} \ \in\ \omega \Rightarrow \omega \Rightarrow \omega$
- $\mathsf{map} \ \in\ (\iota \Rightarrow \iota) \Rightarrow \omega \Rightarrow \omega$
- $\mathsf{restrict} \ \in\ \tau \Rightarrow \omega \Rightarrow \omega$

Proof: The meta-types are justified by the theorems of §A.2. For example, $\mathsf{lex}(R, S)$ is defined as the lexicographic ordering of the orderings R and S. It is well-founded over $\iota_=$ if R and S are well-founded over $\iota_=$, and therefore has the meta-type $\omega \Rightarrow \omega \Rightarrow \omega$. ■

$$\begin{array}{rcl}
a \prec_{\mathsf{pR}} b & \leftrightarrow & (b = \mathsf{s}(a)) \ \vee \ (\exists\, x.\ b = \langle a, x\rangle) \ \vee \\
 & & \quad (\exists\, x.\ b = \langle x, a\rangle) \ \vee \ (\exists\, x.\ a \prec_{\mathsf{pR}} x \wedge x \prec_{\mathsf{pR}} b) \\
\langle a_1, a_2\rangle \prec_{\mathsf{lex}(R,S)} \langle b_1, b_2\rangle & \leftrightarrow & (a_1 \prec_R b_1) \ \vee \ (a_1 = b_1 \wedge a_2 \prec_S b_2) \\
a \prec_{\mathsf{map}(f,R)} b & \leftrightarrow & f(a) \prec_R f(b) \\
a \prec_{\mathsf{restrict}(A,R)} b & \leftrightarrow & a : A \ \wedge \ b : A \ \wedge \ a \prec_R b
\end{array}$$

Figure 3.4: *Rules for Well-Founded Orderings in CCL*

Theorem 3.19 *The bi-implications in Figure 3.4 are derivable.*

As well-founded orderings are defined over equivalences classes ($\iota_=$), the following theorem holds for substitution.

Theorem 3.20 *For all terms a, a', b and b' and well-founded orderings R, if $a = a'$ and $b = b'$, then $a \prec_R b$ iff $a' \prec_R b'$.*

3.6 The Theory CCL

The computational logic CCL is based on classical first-order logic. In addition, meta-types are defined for terms ($\iota_=$), types (τ) and well-founded orderings (ω), and predicates are defined for term equality ($=$), type equality ($=$), type inhabitation ($:$), monotonicity (*Mono*) and well-founded orderings ($\prec$).

First-Order Logic

A natural deduction calculus for first-order logic is used, with introduction and elimination rules for the constants $\wedge$, $\vee$, $\supset$,$\forall$ and $\exists$, an elimination rule for $\bot$ and the rule *classical* to make the calculus classical (Figure 3.5). Quantification is over the meta-type of terms ($\iota_=$) and the sort of types (τ). The remaining constants are defined by

$$\begin{array}{rcl}
\neg P & \equiv & P \supset \bot \\
\top & \equiv & \neg\bot \\
P \leftrightarrow Q & \equiv & (P \supset Q) \ \wedge \ (Q \supset P)
\end{array}$$

Typed quantification is defined for terms by

$$\begin{array}{rcl}
\forall\, x{:}A.P(x) & \equiv & \forall\, x.\ x : A \supset P(x) \\
\exists\, x{:}A.P(x) & \equiv & \exists\, x.\ x : A \wedge P(x)
\end{array}$$

Equality

The predicate $=$ is used for equality over both terms and types.

$$\begin{array}{rcl}
= & \in & \iota_= \Rightarrow \iota_= \Rightarrow o \\
= & \in & \tau \Rightarrow \tau \Rightarrow o
\end{array}$$

$$\wedge\text{-}\mathit{intr} \quad \frac{P \qquad Q}{P \wedge Q} \qquad\qquad \wedge\text{-}\mathit{elim} \quad \frac{P \wedge Q}{P} \quad \frac{P \wedge Q}{Q}$$

$$\vee\text{-}\mathit{intr} \quad \frac{P}{P \vee Q} \quad \frac{Q}{P \vee Q} \qquad\qquad \vee\text{-}\mathit{elim} \quad \frac{P \vee Q \qquad \begin{matrix}\left[\, P \,\right] \\ R\end{matrix} \qquad \begin{matrix}\left[\, Q \,\right] \\ R\end{matrix}}{R}$$

$$\supset\text{-}\mathit{intr} \quad \frac{\begin{matrix}\left[\, P \,\right] \\ Q\end{matrix}}{P \supset Q} \qquad\qquad \supset\text{-}\mathit{elim} \quad \frac{P \qquad P \supset Q}{Q}$$

$$\forall\text{-}\mathit{intr} \quad \frac{\overline{P(x)}^{\,x}}{\forall x.\ P(x)} \qquad\qquad \forall\text{-}\mathit{elim} \quad \frac{\forall x.\ P(x)}{P(t)}$$

$$\exists\text{-}\mathit{intr} \quad \frac{P(t)}{\exists x.\ P(x)} \qquad\qquad \exists\text{-}\mathit{elim} \quad \frac{\exists x.\ P(x) \qquad \begin{matrix}\left[\, P(x) \,\right]_x \\ Q\end{matrix}}{Q}$$

$$\bot\text{-}\mathit{elim} \quad \frac{\bot}{P} \qquad\qquad \mathit{classical} \quad \frac{\begin{matrix}\left[\, \neg P \,\right] \\ P\end{matrix}}{P}$$

Figure 3.5: *Natural Deduction Rules for First-Order Logic*

$$a = a \qquad \frac{a = b}{b = a}$$

$$\frac{a = b \qquad b = c}{a = c} \qquad \frac{a = b \qquad P(a)}{P(b)}$$

Figure 3.6: *General Rules of Equality for CCL*

$$\begin{array}{rcl}
\mathsf{scase}(\mathsf{z}, b, c) & = & b \\
\mathsf{scase}(\mathsf{s}(a), b, c) & = & c(a) \\
\mathsf{pcase}(\mathsf{lam}\ x.b(x), c) & = & c(b) \\
\mathsf{split}(\langle a, b\rangle, c) & = & c(a, b)
\end{array}
\qquad
\begin{array}{c}
\mathsf{rec}(a, h) \quad = \quad h(a, \lambda x.\mathsf{rec}(x, h)) \\
\dfrac{a : A}{\mathsf{let}\ x\ \mathsf{be}\ a\ \mathsf{in}\ b(x)\ \mathsf{end} \quad = \quad b(a)}
\end{array}$$

Figure 3.7: *Conversion Rules for CCL*

Figure 3.6 presents general rules for reasoning about equality. For terms, the rules for reflexivity, symmetry and transitivity are justified by Theorem 3.5, and for types by the properties of set equality. Equal terms may be substituted in terms (Theorem 3.7), in types and in predicates. As type equality, which is just set equality, is also a congruence, a general rule of substitution is admissible.

$$\frac{a = b \qquad P(a)}{P(b)}$$

The conversion rules for terms and, where appropriate, types (Figure 3.7) immediately follow from Theorems 3.8 and 3.13. The congruence rules for terms (Figure 3.8) follow from Theorem 3.7.

3.7 The Strength of CCL

CCL allows reasoning about the total correctness of programs of $\mathcal{L}$. Although the programming language $\mathcal{L}$ is clearly Turing complete (it contains untyped λ-abstraction), there is, as yet, no characterisation of the class of programs that can be typed in CCL. Although all programs that can be typed in CCL terminate, it is not the case that all terminating programs can be typed—consider, for example, the program $\mathsf{s}(\Omega)$. In fact, the typeable programs correspond more closely to Martin-Löf's notion of *hereditarily terminating* programs [40]. But the class of typeable programs depends on the set of well-founded orderings that can be constructed. The constructors for well-founded orderings are not intended to be complete, although they are sufficient for the programs described in this dissertation.

$$\frac{a = a'}{\mathsf{s}(a) = \mathsf{s}(a')} \qquad \frac{\overline{b(x) = b'(x)}^{\,x}}{\mathsf{lam}\ x.b(x) = \mathsf{lam}\ x.b'(x)}$$

$$\frac{a = a' \qquad b = b'}{\langle a, b\rangle = \langle a', b'\rangle} \qquad \frac{a = a' \qquad b = b' \qquad \overline{c(x) = c'(x)}^{\,x}}{\mathsf{scase}(a, b, c) = \mathsf{scase}(a', b', c')}$$

$$\frac{f = f' \qquad \overline{c(u) = c'(u)}^{\,u}}{\mathsf{pcase}(f, c) = \mathsf{pcase}(f', c')} \qquad \frac{p = p' \qquad \overline{c(x, y) = c'(x, y)}^{\,x,y}}{\mathsf{split}(p, c) = \mathsf{split}(p', c')}$$

$$\frac{a = a' \qquad \overline{h(x, u) = h'(x, u)}^{\,x,u}}{\mathsf{rec}(a, h) = \mathsf{rec}(a', h')} \qquad \frac{a = a' \qquad \overline{b(x) = b'(x)}^{\,x}}{\mathsf{let}\ x\ \mathsf{be}\ a\ \mathsf{in}\ b(x)\ \mathsf{end} = \mathsf{let}\ x\ \mathsf{be}\ a'\ \mathsf{in}\ b'(x)\ \mathsf{end}}$$

Figure 3.8: *Congruence Rules for CCL*

For example, multi-set orderings cannot be constructed, and so an algorithm for proof normalisation cannot be typed. A new constructor for multi-set orderings could, of course, be added.

A result is proved that puts a lower bound on the class of typeable programs, showing that any program that is feasibly computable, from a complexity viewpoint, can be typed in CCL. The result is of limited value; it shows what can be represented, but says nothing about how this is done. In fact, the proof relies only on primitive recursion being typeable in CCL.

Theorem 3.21 *All those functions that are provably total in Peano Arithmetic(PA) can be represented as typed terms in CCL.*

Proof: It is known that the closed terms of type $Int \to Int$ in Gödel's system T are precisely those that are provably total in PA [19]. There is an embedding of system T in CCL, for which typeable functions in system T are mapped to typeable programs in CCL. Therefore, CCL can type at least the functions provably total in PA.

The calculus for Gödel's system T is given in Figure 3.9, and its embedding in CCL is given below. A term t of system T is represented by the term t° of CCL.

$$\begin{array}{lcl}
(lam\ x\,.\,b(x))^\circ & \equiv & \mathsf{lam}\ x.b(x)^\circ \\
\langle a, b\rangle^\circ & \equiv & \langle a^\circ, b^\circ\rangle \\
T^\circ & \equiv & \mathsf{z} \\
F^\circ & \equiv & \mathsf{s(z)} \\
0^\circ & \equiv & \langle \mathsf{z}, \mathsf{z}\rangle \\
S(n)^\circ & \equiv & \langle \mathsf{s(z)}, n^\circ\rangle \\
 & & \\
(f\ \text{‘}\ a)^\circ & \equiv & f^\circ\ \text{‘}\ a^\circ \\
I(p)^\circ & \equiv & \mathsf{split}(p^\circ, \lambda x\ y.x)
\end{array}$$

$$\begin{array}{lll}
J(p)^\circ & \equiv & \mathsf{split}(p^\circ, \lambda x\ y.y) \\
D^\circ & \equiv & \mathsf{lam}\ t.\mathsf{lam}\ u.\mathsf{lam}\ b.\mathsf{scase}(b, t, \lambda x.u) \\
R^\circ & \equiv & \mathsf{lam}\ t.\mathsf{lam}\ u.\mathsf{lam}\ n.\mathsf{rec}(n, \\
 & & \qquad\qquad \lambda x\ g.\mathsf{split}(x, \lambda y\ z.\mathsf{scase}(y, t, \lambda x.u \text{ ‘ } g(x) \text{ ‘ } x)))
\end{array}$$

Type A of system T is represented by the type A° of CCL.

$$\begin{array}{lll}
(A \to B)^\circ & \equiv & \mathsf{\Pi} x{:}A^\circ.B^\circ \\
(A \times B)^\circ & \equiv & \Sigma x{:}A^\circ.B^\circ \\
\mathit{Bool}^\circ & \equiv & \mathsf{S}(\mathsf{S}(\mathsf{V})) \\
\mathit{Int}^\circ & \equiv & \mu\, X.\, \Sigma x{:}\mathsf{S}(\mathsf{S}(\mathsf{V})).\mathsf{scase}(x, \mathsf{S}(\mathsf{V}), \lambda x.X)
\end{array}$$

The predicates : and $\rightsquigarrow$ are represented by predicates in CCL.

$$\begin{array}{lll}
(a : A)^\circ & \equiv & a^\circ : A^\circ \\
(t \rightsquigarrow a)^\circ & \equiv & t^\circ = a^\circ
\end{array}$$

With these definitions, it is simple to prove that the rules of Figure 3.9 map to theorems of CCL. ■

lam	$`$	$\to$
$\langle\rangle$	I, J	$\times$
T, F	D	Bool
$0, S$	R	Int

Constants

$$\frac{\begin{matrix}\left[\ x : A\ \right]_x \\ b(x) : B\end{matrix}}{\mathit{lam}\ x\,.\,b(x) : A \to B} \qquad \frac{f : A \to B \qquad a : A}{f\ `\ a : B}$$

$$\frac{a : A \qquad b : B}{\langle a, b\rangle : A \times B} \qquad \frac{p : A \times B}{I(p) : A} \qquad \frac{p : A \times B}{J(p) : B}$$

$$T : \mathit{Bool} \qquad F : \mathit{Bool} \qquad \frac{t : A \qquad u : A \qquad b : \mathit{Bool}}{D\ `\ t\ `\ u\ `\ b : A}$$

$$0 : \mathit{Int} \qquad \frac{n : \mathit{Int}}{S(n) : \mathit{Int}} \qquad \frac{t : A \qquad u : A \to (\mathit{Int} \to A) \qquad n : \mathit{Int}}{R\ `\ t\ `\ u\ `\ n : A}$$

Type Rules

$$(\mathit{lam}\ x\,.\,b(x))\ `\ a \ \rightsquigarrow\ b(a)$$

$$D\ `\ u\ `\ v\ `\ T \ \rightsquigarrow\ u \qquad R\ `\ u\ `\ v\ `\ 0 \ \rightsquigarrow\ b(a)$$

$$D\ `\ u\ `\ v\ `\ F \ \rightsquigarrow\ v \qquad R\ `\ u\ `\ v\ `\ S(t) \ \rightsquigarrow\ v\ `\ (R\ `\ u\ `\ v\ `\ t)\ `\ t$$

Conversion Rules

Figure 3.9: *Gödel's System T*

Chapter 4

Implementation

This chapter describes the implementation of CCL and the development of a system for proving program correctness using the generic theorem prover Isabelle [55].

The natural deduction calculus for CCL presented in Chapter 3 is implemented in Isabelle. The basic rules provide inductive reasoning over the structure of programs and partial evaluation of terms; rules are also derived within CCL for more general forms of induction and the freeness of constructors. In addition, common programming data types are encoded using the primitive type-formers of CCL (V, S, Π, Σ and μ). There is potential, as yet unrealised, to define new data types and automatically derive facts about them from descriptions resembling `datatype` statements in ML.

Typing a program in CCL ensures its termination (Theorem 3.10). Moreover, using dependent type-formers and subtypes allows typing to ensure more fine grain behaviour (program correctness). Thus, CCL offers a framework in which to generate *correctness conditions* sufficient for a program to meet a specification. Tactics are developed for reasoning about program correctness, including type checking and rewriting. But explicit typing, which allows the encoding of general data types and the convenient specification of programs, makes the logic clumsy for proving facts within particular domains. Instead, lemmas can be proved in first-order logic extended with computational types, and then lifted up to CCL. This means sometimes choosing to work in a subtheory of CCL, in which typing is decidable and well known techniques for reasoning by induction and rewriting are applicable.

§4.1 presents an overview of the theorem prover Isabelle and describes the implementation of the basic rules of CCL. Additional rules are derived in §4.2. Common data types and constants are defined and facts proved about them in §4.3. §4.4 describes the tactics used for reasoning about program correctness. §4.5 describes how first-order logic is interpreted in CCL and used to prove lemmas that arise in proofs of program correctness. Finally, §4.6 summarises the complete implementation.

4.1 Encoding CCL in Isabelle

This section introduces the theorem prover Isabelle and describes the implementation of the basic rules of CCL.

4.1.1 Overview of Isabelle

Isabelle is a generic theorem prover (logical framework) written in ML that supports reasoning in *object-logics* by encoding them as natural deduction calculi in its *meta-logic*. The meta-logic is a fragment of intuitionistic higher-order logic including implication $\Longrightarrow$, universal quantification $\bigwedge$, equality $\equiv$ and abstraction λ. Types of the meta-logic, *meta-types*, are classified into *sorts*, which are partially ordered by an inclusion relation $\Subset$ such that $s_1 \Subset s_2$ iff every element of s_1 is an element of s_2. Meta-type variables are restricted to range over elements of a particular sort, which provides a form of *order-sorted polymorphism* (cf. Nipkow [46] and the language Haskell [29]). Write α_s for a variable ranging over meta-types of sort s.

Isabelle's basic meta-logic has two sorts: $\mathit{logic} \Subset \mathit{any}$. Meta-types are constructed from the meta-type of propositions *prop*, of sort *logic*, and the meta-type former of meta-functions $\Rightarrow$, which takes two meta-types of sort *logic* (or *any*) and produces a meta-type of sort *logic* (or *any*). The infix predicate :: denotes both meta-type membership and sort membership. Constants of the meta-logic have the following meta-types

$$\begin{aligned} \bigwedge &:: (\alpha_{\mathit{logic}} \Rightarrow \mathit{prop}) \Rightarrow \mathit{prop} \\ \Longrightarrow &:: \mathit{prop} \Rightarrow \mathit{prop} \Rightarrow \mathit{prop} \\ \equiv &:: \alpha_{\mathit{logic}} \Rightarrow \alpha_{\mathit{logic}} \Rightarrow \mathit{prop} \end{aligned}$$

Additional sorts, meta-types and constants may be defined to encode new object-logics. Isabelle imposes restrictions on the possible orderings of sorts to ensure that a 'principal type property' is retained for the Hindley-Milner style of type checking used in the meta-logic [14, 27]. Order-sorted polymorphism allows object-logic constants to be polymorphic—consider, for example, the equality relation of some first-order logics. Meta-level abstraction allows object-logics to be defined with the higher-order syntax described in §3.1. A pretty printer and parser provide concrete syntax.

The rule calculus of an object-logic is encoded as a set of axioms in the meta-logic. For each form of judgement in the object-logic, a meta-function *istrue* is defined to lift elements of that judgement up to propositions in the meta-logic. This allows $\Longrightarrow$ to represent entailment, $\bigwedge$ to bind variables and $\equiv$ to encode definitions. Write $[\![\, H1;\ H2;\ \ldots \,]\!] \Longrightarrow P$ for the sequence of hypotheses $H1 \Longrightarrow H2 \Longrightarrow \ldots \Longrightarrow P$. For example, Isabelle's first-order logic `FOL` has a single judgement of the truth of formulae (of meta-type o), for which the meta-function $\mathit{istrue} :: o \Rightarrow \mathit{prop}$ is defined. Using this function, the $\wedge$-*intr* rule

$$\frac{P \qquad Q}{P \wedge Q}$$

is encoded as

$$\bigwedge P . \bigwedge Q . [\![\, \mathit{istrue}(P);\ \mathit{istrue}(Q) \,]\!] \Longrightarrow \mathit{istrue}(P \wedge Q)$$

which captures the intended meaning that for all formulae P and Q, if P is true and if Q is true then $P \wedge Q$ is true. Meta-level quantification can be used to bind variables as well as quantify over formulae (in the meta-logic they are both just terms). Writing $\forall(\lambda x . P(x))$ as $\forall x . P(x)$, the $\forall$-*intr* rule

$$\frac{\overline{P(x)}^{\,x}}{\forall x . P(x)}$$

is encoded as

$$\bigwedge P . (\bigwedge x . \mathit{istrue}(P(x))) \Longrightarrow \mathit{istrue}(\forall x . P(x))$$

which captures the intended meaning that for all predicates P, if $P(x)$ is true for all x then $\forall x.\, P(x)$ is true. The eigenvariable condition is enforced by binding x within the hypothesis.

In Isabelle, *proof rules* are a generalised form of Horn clause. *Axioms* of an object-logic are used as proof rules. Derived *theorems*, having the same form as axioms, are used as proof rules with no extra cost. Isabelle supports *logical variables*, indicated by a leading question mark $?x$. As in Prolog, these may appear in goals and become instantiated incrementally during proofs.

Proofs may be built up from rules in both a forward and backward direction. Stepping backward in a proof is by *resolving* a rule (an axiom or a theorem) with one of the current goals: variables quantified at the outermost level of the rule are replaced by logical variables; then the conclusion of the rule is unified with the selected goal, and on successful unification this goal is replaced by the instantiated premises. Higher-order unification [30] handles the inherent $\alpha\beta\eta$-conversions. The procedure is incomplete and may yield an infinite set of unifiers. But in practice this rarely gives rise to problems; when it does, additional tactics can be employed to guide instantiation. Each step in a backward proof replaces one of the current goals by zero or more new goals; when no subgoals remain, the proof is complete. Forward reasoning is less well supported; rules may be 'glued' together by unifying the conclusion of one rule with a premise of another.

Individual steps in a proof are made by *tactics*. For example, `assume_tac n` solves subgoal `n` by assumption, and `resolve_tac rls n` uses resolution on subgoal `n` with the first rule in `rls` whose conclusion unifies with the subgoal. *Tacticals* allow tactics to be combined. For example, sequentially using tac_1 `THEN` tac_2, as alternatives using tac_1 `ORELSE` tac_2 and repeatedly using `REPEAT` *tac*. More complex tacticals are coded in the programming language ML.

As `resolve_tac` uses a list of possible rules each of which may yield many results, it returns a possibly infinite stream of new proof states. Together with some non-deterministic tacticals, these produce alternatives in the proof tree that are maintained by Isabelle to allow backtracking, and with appropriate tacticals proof search. Prolog exhibits one possible search strategy, and in its pure form may be implemented as an Isabelle tactical.

Isabelle has a suite of tactics (including `fast_tac`) for automatically solving goals in first-order logic by natural deduction inference. These tactics use a set of introduction and elimination rules for first-order logic (`FOL_cs`) to which further rules can be added. In addition, Isabelle has a generic rewriting package for object-logics, the *simplifier*. Supplied with the basic theorems necessary to justify rewriting (namely reflexivity and transitivity) together with reductions and rules for case analysis, it returns a suite of simplification tactics. The simplifier may rewrite using several reduction relations at once, for example, term equality ($=$) and logical equivalence ($\leftrightarrow$) in first-order logic. Rewriting and congruence rules for the reduction relations are maintained in *simplification sets*. The simplification tactics provide various combinations of the following: rewriting using rules of the simplification set, rewriting using the assumptions in a goal and case analysis using split rules (e.g. for *if* _ *then* _ *else* _). In particular

- `ASM_SIMP_TAC` rewrites a goal using the set of simplification rules and any assumptions, and

- `ASM_SIMP_CASE_TAC` rewrites as `ASM_SIMP_TAC` and also uses case analysis.

4.1.2 The Encoding of CCL

With the exception of hypothetical hypotheses, the basic rules of CCL are directly encoded in Isabelle. The implementation `CCL` is an extension of Isabelle's theory `FOL`, which is an implementation of typed first-order logic. Isabelle encodings are presented in `typewriter` font. Those symbols for which there is no transliteration are translated as follows:

λ	*becomes*	`%`	$\bigwedge$	*becomes*	`!!`
$\equiv$	*becomes*	`==`	α_s	*becomes*	`'a::s`

$a \prec_R b$ *becomes* `[a R b]`

The Isabelle theory `FOL` is briefly introduced before the extension `CCL`.

FOL

`FOL` has only one meta-type `o` of sort `logic`, representing formulae. There is a single form of judgement encoded by the function `istrue` (see §4.1.1).

```
istrue :: o => prop
```

It is not explicitly shown again. The new sort `term` is introduced with the ordering

`term` $\Subset$ `logic` $\Subset$ `any`

which distinguishes meta-types of individuals from formulae, so that first-order quantification can be expressed.

The propositional connectives $\wedge,\vee,\neg,\supset$ and $\leftrightarrow$ are represented by the infix and prefix constants `&`,`|`,`~`,`-->` and `<->` respectively, defined as meta-functions over the meta-type of formulae `o`. As the logic is first-order, quantification is restricted to individuals (the quantifiers are polymorphic over types of sort `term`).

```
All    :: ('a::term => o) => o
Exists :: ('a::term => o) => o
```

Concrete syntax provides

`ALL x.P(x)`	*in place of*	`All(%x.P(x))`
`EX x.P(x)`	*in place of*	`Exists(%x.P(x))`

The axioms of `FOL` are exactly the logical part of CCL (Figure 3.5).

CCL

`CCL` is an extension of `FOL`. Formulae of `CCL` correspond to formulae in `FOL`. Other syntactic classes are represented by the following new meta-types.

`i :: term`	*for the terms of* $\mathcal{L}$ ($\iota_=$)
`t :: term`	*for the types of* $\mathcal{L}$ (τ)
`w :: logic`	*for the well-founded relations* (ω)

The higher-order syntax used in CCL (see §3.1) translates directly into Isabelle's meta-logic. At the level of concrete syntax, alternatives are provided for Π and Σ in the cases when they are truly dependent or not.

```
PROD x:A.B(x)  in place of  Pi(A,%x.B(x))
A -> B         in place of  Pi(A,%x.B)
SUM x:A.B(x)   in place of  Sigma(A,%x.B(x))
A * B          in place of  Sigma(A,%x.B)
```

The pretty printer is sophisticated enough to use these forms where appropriate.

At present, Isabelle does not provide tactics for handling subgoals with nested meta-implication, and so rules with hypothetical hypotheses cannot be used. Instead, the meta-level universal quantification and implication used to represent hypothetical hypotheses are replaced by universal quantification and implication in CCL. This is justified as the meta-logic is higher-order logic and the connectives of CCL are those of first-order logic. For example, the rule rec-*type*

$$\dfrac{a : A \qquad \begin{matrix}\left[\begin{matrix} & [\, y : A \,;\ y \prec_R x \,]_y \\ x : A\,; & u(y) : B(y)\end{matrix}\right]_{x,u} \\ h(x,u) : B(x)\end{matrix}}{\mathsf{rec}(a,h) : B(a)}$$

is replaced by the rule

$$\dfrac{a : A \qquad \begin{matrix}\left[\ x : A\,;\ \forall y{:}A. y \prec_R x \supset u(y) : B(y)\ \right]_{x,u} \\ h(x,u) : B(x)\end{matrix}}{\mathsf{rec}(a,h) : B(a)}$$

For convenience, rules are still written with hypothetical hypotheses, even though the Isabelle implementation uses object-level quantification and implication.

The rules of CCL, modified to remove hypothetical hypotheses, are the only axioms postulated in the Isabelle implementation. Further rules are all derived within the Isabelle theory `CCL`.

4.2 Derived Rules

From the basic rules of CCL, additional rules are derived within the theory for reasoning about program correctness. In particular, rules are derived for

- induction on the type-formers,
- strengthened type rules for destructors, and
- the freeness of constructors.

4.2.1 Induction Rules

The type rule for each destructor $d \in \mathit{DES}$ provides case analysis on formulae $d(\overline{x}) : A$; rec-*type* provides induction on formulae $\mathsf{rec}(a, h) : B(a)$. For example, the rule scase-*type*

$$\frac{a : \mathsf{S}(A) \qquad b : B(\mathsf{z}) \qquad \begin{matrix}\left[\; x : A \;\right]_x \\ c(x) : B(\mathsf{s}(x))\end{matrix}}{\mathsf{scase}(a, b, c) : B(a)}$$

provides case analysis over the canonical elements of $\mathsf{S}(A)$ only for formulae of the form $\mathsf{scase}(a, b, c) : B(a)$. Within CCL, more general forms of case analysis and induction are derived from these rules using the subtype type-former. From the rule scase-*type*, the following induction rule (scase-*ind*) is derived.

$$\frac{a : \mathsf{S}(A) \qquad P(\mathsf{z}) \qquad \begin{matrix}\left[\; x : A \;\right]_x \\ P(\mathsf{s}(x))\end{matrix}}{P(a)}$$

Derivation: [1]
Assuming $a : \mathsf{S}(A)$, $P(\mathsf{z})$ and $x : A \Longrightarrow P(\mathsf{s}(x))$, prove P(a).
By $\{\}$-*elim*, $\mathsf{scase}(a, \mathsf{z}, \lambda x.\mathsf{z}) : \{x{:}\mathsf{S}(\mathsf{V}), P(a)\}$ implies $P(a)$. Using scase-*type*, this can be reduced to the goals

1. $a : \mathsf{S}(A)$
2. $\mathsf{z} : \{x{:}\mathsf{S}(\mathsf{V}), P(\mathsf{z})\}$
3. $y : A \Longrightarrow \mathsf{z} : \{x{:}\mathsf{S}(\mathsf{V}), P(\mathsf{s}(y))\}$

which follow from the assumptions, $\{\}$-*intr* and z-*type*. ■

There is an induction rule for each destructor. In particular, the rule vcase-*ind* asserts that the type V is uninhabited;

$$\frac{a : \mathsf{V}}{P}$$

the rule pcase-*ind* is an induction(η) rule for lam;

$$\frac{f : \Pi(A, B) \qquad \begin{matrix}\left[\begin{matrix}\left[\, x : A \,\right]_x \\ u(x) : B(x)\end{matrix}\right]_u \\ P(\mathsf{lam}\ x.u(x))\end{matrix}}{P(f)}$$

and the rule rec-*ind* is well-founded induction.

$$\frac{a : A \qquad \begin{matrix}\left[\begin{matrix} & \left[\, y : A \,;\ \ y \prec_R x \,\right]_y \\ x : A\,; & P(y)\end{matrix}\right]_x \\ P(x)\end{matrix}}{P(a)}$$

Note that defining $\mathcal{L}$ with application, instead of the more general destructor for functions pcase, would not have allowed the derivation of an η-rule.

[1] We use the term *derivation* for all proofs carried out using the Isabelle implementation of CCL.

4.2.2 Strong Type Rules

In type rules for destructors, the variable being reduced is replaced in the premises by the appropriate canonical forms. For example, in the rule scase-*type*

$$\frac{a:\mathsf{S}(A) \qquad b:B(\mathsf{z}) \qquad \begin{matrix}\left[\, x:A \,\right]_x \\ c(x):B(\mathsf{s}(x))\end{matrix}}{\mathsf{scase}(a,b,c):B(a)}$$

the variable a in $B(a)$ is replaced by the canonical forms z and $\mathsf{s}(x)$. But these implicit substitutions do not occur in any assumptions that might be present. To overcome this, strengthened rules are derived in which the substitutions explicitly appear as hypotheses in the premises. For example, the strengthened type rule for scase is

$$\frac{a:\mathsf{S}(A) \qquad \begin{matrix}\left[\, a=\mathsf{z} \,\right] \\ b:B(\mathsf{z})\end{matrix} \qquad \begin{matrix}\left[\, x:A\,;\;\; a=\mathsf{s}(x) \,\right]_x \\ c(x):B(\mathsf{s}(x))\end{matrix}}{\mathsf{scase}(a,b,c):B(a)}$$

in which the second and third premises are weakened by additional assumptions.

Derivation:

Assuming $a:\mathsf{S}(A)$, $a=\mathsf{z} \Longrightarrow b:B(\mathsf{z})$ and $[\![\, x:A;\;\; a=\mathsf{s}(x) \,]\!] \Longrightarrow c(x):B(\mathsf{s}(x))$, prove $\mathsf{scase}(a,b,c):B(a)$.

By $\supset$-*intr* and *refl*, $a=a\supset \mathsf{scase}(a,b,c):B(a)$ implies $\mathsf{scase}(a,b,c):B(a)$. This is reduced by scase-*ind* to the goals

1. $a:\mathsf{S}(A)$
2. $a=\mathsf{z}\supset \mathsf{scase}(a,b,c):B(\mathsf{z})$
3. $x:A \Longrightarrow a=\mathsf{s}(a)\supset \mathsf{scase}(a,b,c):B(\mathsf{s}(a))$

Note that not all occurrences of a in $a=a\supset \mathsf{scase}(a,b,c):B(a)$ are replaced. These goals follow from the assumptions, *subst* and the conversion rules for scase. ■

A similar technique is used in Martin-Löf's Type Theory. But in a type theory where proofs are identified with programs, the strengthened rules contain extra detail in their proof terms which is redundant—in CCL, the program fragment $\mathsf{scase}(a,b,c)$ remains unchanged when the rule is strengthened.

Similarly, the type rules for pcase and split are strengthened. But rules involving induction cannot be strengthened in this way. In particular, the strengthened version of rec-*type*

$$\frac{a:A \qquad \begin{matrix}\left[\, x:A\,; \;\; \begin{matrix}[\,y:A\,;\;\; y\prec_R x\,]_y \\ u(y):B(y)\end{matrix} \;\; ;\;\; a=x \,\right]_{x,u} \\ h(x,u):B(x)\end{matrix}}{\mathsf{rec}(a,h):B(a)}$$

is inconsistent in CCL, giving further motivation for the separation of case analysis and recursion. This is clearer for the more familiar case of mathematical induction. The rule

$$\frac{\begin{matrix}\left[\, n=0 \,\right] \\ P(0)\end{matrix} \qquad \begin{matrix}\left[\, P(x)\,;\;\; n=x+1 \,\right]_x \\ P(x+1)\end{matrix}}{P(n)}$$

allows the derivation of the fallacy $n = 2 \Longrightarrow n = 0$. An instance of this rule, in which $P(n)$ is $n = 0$, reduces the fallacy to the following subgoals.

- $[\![\, n = 2;\ n = 0 \,]\!] \Longrightarrow 0 = 0$
- $[\![\, n = 2;\ x = 0;\ n = x + 1 \,]\!] \Longrightarrow x + 1 = 0$

The first is trivial; the second has assumptions that lead to the contradiction $2 = 1$, and so it holds as well.

4.2.3 Freeness of Constructors

Freeness of constructors is essential for reasoning about terms in CCL. Distinctness and injectivity of the constructors are derivable within CCL.

In Martin-Löf's Type Theory, constructors are shown to be distinct by the differing reductions they induce in their destructor. But to show that these reductions are different, they must reduce to terms that are distinguishable within the logic. The only examples of this are in the universe of types, namely encodings of the empty type and a non-empty type [62]. Similarly in CCL, the terms z and $\mathsf{s}(a)$ are distinguished by their effect when used as arguments in $\lambda x\mathsf{scase}(x, \mathsf{V}, \lambda x\mathsf{S}(\mathsf{V}))$. In this case, the following rule can be derived.

$$\frac{\mathsf{z} = \mathsf{s}(a)}{P}$$

Derivation:
Assuming $\mathsf{z} = \mathsf{s}(a)$, prove P.
By the conversion rules for scase and scase-*ind*, $\mathsf{z} : \mathsf{scase}(\mathsf{z}, \mathsf{V}, \lambda x.\mathsf{S}(\mathsf{V}))$ implies P. Substituting $\mathsf{s}(a)$ for z, by assumption, reduces this to $\mathsf{z} : \mathsf{scase}(\mathsf{s}(a), \mathsf{V}, \lambda x.\mathsf{S}(\mathsf{V}))$, which follows from the conversion rules for scase and z-*type*. ■

The injectivity of constructors follows from their behaviour in destructors that project their arguments (e.g. $\mathsf{scase}(\mathsf{s}(a), b, \lambda x.x)$ for the constructor s). For example, the following rule can be derived

$$\frac{\mathsf{s}(a) = \mathsf{s}(a')}{a = a'}$$

Derivation:
Assuming $\mathsf{s}(a) = \mathsf{s}(a')$, prove $a = a'$.
By the conversion rules for scase, $\mathsf{scase}(\mathsf{s}(a), b, \lambda x.x) = \mathsf{scase}(\mathsf{s}(a'), b, \lambda x.x)$ implies that $a = a'$. Substituting $\mathsf{s}(a)$ for $\mathsf{s}(a')$ reduces this to $\mathsf{scase}(\mathsf{s}(a), b, \lambda x.x) = \mathsf{scase}(\mathsf{s}(a), b, \lambda x.x)$ which follows by *refl*. ■

Similarly, pairing is injective.

$$\frac{\langle a, b\rangle = \langle a', b'\rangle \qquad \begin{matrix}\left[\ a = a'\ ;\ \ b = b'\ \right] \\ P\end{matrix}}{P}$$

The constructor lam is also injective, though this is of little use in practice.

4.3 Definitions

CCL contains a core programming language $\mathcal{L}$. This is enriched with definitions for locally declared functions and data types from which type and conversion rules are derived.

Executing a program in this enriched language uses derived evaluation rules to supplement those of $\mathcal{L}$. The derived rules just provide short cuts in the evaluation of programs—the same result would be achieved by unfolding the definitions before execution. They do not add any expressive power to $\mathcal{L}$, nor do they compromise the properties proved for evaluation.

4.3.1 Local Declarations

Two sets of constants are defined for introducing auxiliary functions that are non-recursive and recursive. Non-recursive functions of one or more arguments are introduced with the term-formers

$$\begin{array}{rcl} \mathsf{letfun} & :: & (\iota \Rightarrow \iota) \Rightarrow (\iota \Rightarrow \iota) \Rightarrow \iota \\ \mathsf{letfun} & :: & (\iota \Rightarrow \iota \Rightarrow \iota) \Rightarrow (\iota \Rightarrow \iota) \Rightarrow \iota \\ & \vdots & \end{array}$$

and recursive functions of one or more arguments with the term-formers

$$\begin{array}{rcl} \mathsf{letrec} & :: & (\iota \Rightarrow \iota \Rightarrow \iota) \Rightarrow (\iota \Rightarrow \iota) \Rightarrow \iota \\ \mathsf{letrec} & :: & (\iota \Rightarrow \iota \Rightarrow \iota \Rightarrow \iota) \Rightarrow (\iota \Rightarrow \iota) \Rightarrow \iota \\ & \vdots & \end{array}$$

Isabelle is able to resolve the overloadings that arise.

The unary cases are sufficient illustration. Write the term $\mathsf{letfun}(\lambda x.a(x), \lambda f.b(f))$ as $\mathsf{letfun} f\, x\ \mathsf{be}\ a(x)\ \mathsf{in}\ b(f)\ \mathsf{end}$, and similarly write the term $\mathsf{letrec}(\lambda x\ g.a(x,g), \lambda f.b(f))$ as $\mathsf{letrec} f\, x\ \mathsf{be}\ a(x,f)\ \mathsf{in}\ b(f)\ \mathsf{end}$. The term-former letfun is defined by

$$\mathsf{letfun} f\, x\ \mathsf{be}\ a(x)\ \mathsf{in}\ b(f)\ \mathsf{end} \quad \equiv \quad b(\mathsf{lam}\ x.a(x))$$

from which the following type and conversion rules are derived.

$$\frac{\begin{array}{cc} \left[\ x : A\ \right]_x & \left[\begin{array}{c} [\,x : A\,]_x \\ v \,`\, x : B(x) \end{array}\right]_v \\ a(x) : B(x) & b(v) : C \end{array}}{\mathsf{letfun} f\, x\ \mathsf{be}\ a(x)\ \mathsf{in}\ b(f)\ \mathsf{end} : C}$$

$$\mathsf{letfun} f\, x\ \mathsf{be}\ a(x)\ \mathsf{in}\ b(f)\ \mathsf{end} = b(\mathsf{lam}\ x.a(x))$$

Note that this definition of letfun has the same meaning as using let to declare functions.

$$\mathsf{letfun} f\, x\ \mathsf{be}\ a(x)\ \mathsf{in}\ b(f)\ \mathsf{end} \quad \equiv \quad \mathsf{let}\ f\ \mathsf{be}\ \mathsf{lam}\ x.a(x)\ \mathsf{in}\ b(f)\ \mathsf{end}$$

But this is not derivable as an equality in CCL, because the conversion rule for let uses typeability to ensure termination, and typeability is stronger than the termination predicate $\downarrow$, which was not formalised in CCL.

Recursive functions of one argument are introduced with letrec, defined by

$$\mathsf{letrec} f\, x\ \mathsf{be}\ a(x,f)\ \mathsf{in}\ b(f)\ \mathsf{end} \quad \equiv \quad \mathsf{letfun} f\, x\ \mathsf{be}\ \mathsf{rec}(x, \lambda u\ g.a(u, \mathsf{lam}\ v.g(v)))\ \mathsf{in}\ b(f)\ \mathsf{end}$$

from which the following type and conversion rules are derived.

$$\dfrac{\begin{array}{c}\left[\begin{array}{cc} & [\,y : A\,;\ \ y \prec_R x\,]_y \\ x : A\,; & u\,\text{‘}\,y : B(y)\end{array}\right]_{x,u} \\ a(x,u) : B(x)\end{array} \qquad \begin{array}{c}\left[\begin{array}{c}[\,x : A\,]_x \\ v\,\text{‘}\,x : B(x)\end{array}\right]_v \\ b(v) : C\end{array}}{\mathsf{letrec} f\, x\ \mathsf{be}\ a(x,f)\ \mathsf{in}\ b(f)\ \mathsf{end} : C}$$

$$\mathsf{letrec} f\, x\ \mathsf{be}\ a(x,f)\ \mathsf{in}\ b(f)\ \mathsf{end} = b(\mathsf{lam}\ x.\mathsf{rec}(x, \lambda x\ g.a(x, \mathsf{lam}\ x.g(x))))$$

The type rule differs from that for letfun only in the presence of an induction hypothesis.

4.3.2 Data Types

Using the basic type-formers of CCL, there are straightforward encodings of the following types: unit, boolean, disjoint union, natural numbers and lists. From these encodings, rules are derived for

- typing the term-formers,
- computation and congruence,
- the freeness of constructors,
- induction,

and, where appropriate,

- functionals for primitive recursion.

Finite enumerations (Unit and Bool) are encoded using the type-formers V and S (Figures 4.1 and 4.2). The term ucase(a, b) is encoded as the term scase$(a, b, \lambda x.\mathsf{vcase})$. For well-typed terms, $\lambda x.\mathsf{vcase}$ is never executed. The same rules would be derivable if vcase was replaced by any other term; using vcase just indicates that this program fragment must be unused. If the typing judgement $a : A$ was separated from other formulae then the use of vcase might be made mandatory; but, as this would prevent typing judgements appearing freely in formulae, the price is too great. Since scase is polymorphic in its result type, ucase and cond will also be polymorphic in their result types—they are defined as destructors that return terms and as destructors that return types.

The disjoint union $A + B$ is encoded as a tagged record using the dependent type-former Σ and the destructor cond (Figure 4.3). Once disjoint union is defined, other types, including inductive ones, may be introduced in a similar fashion to ML's `datatype` statement. If they were not primitive, the types of natural numbers and lists would be defined in ML as

```
datatype     Nat = zero | succ of Nat;
datatype 'a List = nil  | :: of 'a * 'a List;
```

In CCL, their encodings are similar (Figures 4.4 and 4.5).

$$\begin{array}{rcl}\mathsf{Nat} & \equiv & \mu\, X.\, \mathsf{Unit} + X \\ \mathsf{List}(A) & \equiv & \mu\, X.\, \mathsf{Unit} + A \times X\end{array}$$

$$\begin{array}{lcl} \mathsf{Unit} & \equiv & \mathsf{S(V)} \\ \mathsf{one} & \equiv & \mathsf{z} \\ \mathsf{ucase}(a,b) & \equiv & \mathsf{scase}(a,b,\lambda x.\mathsf{vcase}) \end{array}$$

Definitions

$$\mathsf{one} : \mathsf{Unit} \qquad \frac{a : \mathsf{Unit} \qquad b : B(\mathsf{one})}{\mathsf{ucase}(a,b) : B(a)}$$

$$\mathsf{ucase}(\mathsf{one}, b) = b \qquad \frac{a = a' \qquad b = b'}{\mathsf{ucase}(a,b) = \mathsf{ucase}(a',b')}$$

$$\frac{a : \mathsf{Unit} \qquad P(\mathsf{one})}{P(a)}$$

DerivedRules

Figure 4.1: *An Encoding of the Unit Data Type*

Type rules for inductive types are derived using the conversion rule for μ, which requires that the argument to μ is monotonic. For example, the lemma

$$Mono(\lambda X.\mathsf{Unit} + X)$$

immediately follows from the rules for *Mono* and leads to the conversion rule

$$\mu\, X.\, \mathsf{Unit} + X \quad = \quad \mathsf{Unit} \; + \mu\, X.\, \mathsf{Unit} + X$$

which is used to derive the type rules for Nat.

4.4 Tactics

This section introduces tactics for partially automating correctness proofs. These tactics provide type checking and rewriting, introduce recursive functions and local declarations, instantiate induction hypotheses, and solve some goals for well-founded orderings.

4.4.1 Type Checking

Type checking breaks down goals of the form $a : A$ using the type rules of CCL. As there is only one type rule for each term-former, type checking breaks down goals in a unique way. If type A specifies the behaviour of program a, then type checking raises conditions for the correctness of a (including termination).

As CCL has subtypes, solving the goal $a{:}A$ is in general undecidable, though it can be reduced to a set of *correctness conditions* (i.e. formulae not of the form $a{:}A$). *Typeability*, showing that there exists a type A such that $a : A$, is also undecidable, as this may rely on finding subtypes to ensure termination. For example, the following algorithm for subtractive division may not be simply typed as $\mathsf{Nat} \rightarrow \mathsf{Nat} \rightarrow \mathsf{Nat}$, but requires that the divisor be non-zero: $\mathsf{Nat} \rightarrow \{x{:}\mathsf{Nat}, \neg x = \mathsf{zero}\} \rightarrow \mathsf{Nat}$.

$$\begin{array}{lcl} \mathsf{Bool} & \equiv & \mathsf{S(S(V))} \\ \mathsf{true} & \equiv & \mathsf{z} \\ \mathsf{false} & \equiv & \mathsf{s(z)} \\ \mathsf{cond}(b,c,d) & \equiv & \mathsf{scase}(b,c,\lambda x.d) \end{array}$$

DEFINITIONS

$$\mathsf{true} : \mathsf{Bool} \qquad \mathsf{false} : \mathsf{Bool}$$

$$\frac{b : \mathsf{Bool} \qquad c : B(\mathsf{true}) \qquad d : B(\mathsf{false})}{\mathsf{cond}(b,c,d) : B(b)}$$

$$\begin{array}{c} \mathsf{cond}(\mathsf{true},c,d) = c \\ \mathsf{cond}(\mathsf{false},c,d) = d \end{array} \qquad \frac{b = b' \qquad c = c' \qquad d = d'}{\mathsf{cond}(b,c,d) = \mathsf{cond}(b',c',d')}$$

$$\frac{\mathsf{true} = \mathsf{false}}{P} \qquad \frac{b : \mathsf{Bool} \qquad P(\mathsf{true}) \qquad P(\mathsf{false})}{P(b)}$$

DERIVEDRULES

Figure 4.2: *An Encoding of the Boolean Data Type*

$$\begin{array}{lcl} \mathit{divide} & \equiv & \mathsf{lam}\ n.\mathsf{lam}\ d.\ \mathsf{letrec}\ \ \mathit{div}\, n\, d\ \ \mathsf{be}\ \ \mathsf{cond}(\mathit{lt}\,`\,n\,`\,d, \mathsf{zero}, \\ & & \qquad\qquad\qquad\qquad\qquad \mathsf{succ}(\mathit{div}\,`\,(n-d)\,`\,d)) \\ & & \qquad\qquad \mathsf{in}\ \ \mathit{div}\,`\,n\,`\,d\ \ \mathsf{end} \end{array}$$

Tactics are developed for reducing goals of the form $a : A$ to correctness conditions. In addition to the primitive type rules of CCL, strengthened as described in §4.2, these tactics use derived rules for recursive calls and boolean valued functions.

Recursive Calls

The rules for recursive calls (rec-*type* and letrec-*type*) introduce induction hypotheses that use object-level quantification and implication to overcome Isabelle's lack of hypothetical hypotheses. For example,the hypothesis introduced by rec-*type* is

$$\forall y{:}A.y \prec_R x \supset g(y) : B(y)$$

To type check a recursive call, this hypotheses must be broken down using $\forall$-*elim* and $\supset$-*elim*. Rules are derived that implicitly carry out these inferences, so that recursive calls can be type checked by a single rule. For rec, the following rule is derived.

$$\frac{\forall y{:}A.y \prec_R x \supset g(y) : B(y) \qquad \begin{array}{c} \left[\ g(a) : B(a)\ \right] \\ g(a) : D \end{array} \qquad a : A \qquad a \prec_R x}{g(a) : D}$$

Similar rules are derived for calls introduced by letrec.

$$\begin{array}{lcl}
A{+}B & \equiv & \Sigma x{:}\mathsf{Bool}.\mathsf{cond}(x, A, B) \\
\mathsf{inl}(a) & \equiv & \langle \mathsf{true}, a \rangle \\
\mathsf{inr}(b) & \equiv & \langle \mathsf{false}, b \rangle \\
\mathsf{when}(a, c, d) & \equiv & \mathsf{split}(a, \lambda x\ y.\mathsf{cond}(x, c(y), d(y)))
\end{array}$$

Definitions

$$\frac{a : A}{\mathsf{inl}(a) : A{+}B} \qquad \frac{b : B}{\mathsf{inr}(b) : A{+}B}$$

$$\frac{a : A{+}B \qquad \begin{array}{c}\left[\ x : A\ \right]_x \\ c(x) : C(\mathsf{inl}(x))\end{array} \qquad \begin{array}{c}\left[\ y : B\ \right]_y \\ d(y) : C(\mathsf{inr}(y))\end{array}}{\mathsf{when}(a, c, d) : C(a)}$$

$$\begin{array}{c}\mathsf{when}(\mathsf{inl}(a), c, d) = c(a) \\ \mathsf{when}(\mathsf{inr}(a), c, d) = d(a)\end{array} \qquad \frac{a = a' \qquad \overline{c(x) = c'(x)}^{\,x} \qquad \overline{d(y) = d'(y)}^{\,y}}{\mathsf{when}(a, c, d) = \mathsf{when}(a', c', d')}$$

$$\frac{\mathsf{inl}(a) = \mathsf{inr}(b)}{P} \qquad \frac{\mathsf{inl}(a) = \mathsf{inl}(a')}{a = a'} \qquad \frac{\mathsf{inr}(b) = \mathsf{inr}(b')}{b = b'}$$

$$\frac{a : A{+}B \qquad \begin{array}{c}\left[\ x : A\ \right]_x \\ P(\mathsf{inl}(x))\end{array} \qquad \begin{array}{c}\left[\ y : B\ \right]_y \\ P(\mathsf{inr}(y))\end{array}}{P(a)}$$

$$a \prec_{\mathsf{pR}} \mathsf{inl}(a) \qquad\qquad b \prec_{\mathsf{pR}} \mathsf{inr}(b)$$

DerivedRules

Figure 4.3: *An Encoding of the Disjoint Union Data Type*

$$\begin{array}{lcl}
\mathsf{Nat} & \equiv & \mu\, X\,.\,\mathsf{Unit}+X \\
\mathsf{zero} & \equiv & \mathsf{inl}(\mathsf{one}) \\
\mathsf{succ}(n) & \equiv & \mathsf{inr}(n) \\
\mathsf{ncase}(n,b,c) & \equiv & \mathsf{when}(n,\lambda x.b,\lambda y.c(y)) \\
\mathsf{nrec}(n,b,c) & \equiv & \mathsf{rec}(n,\lambda n\ g.\mathsf{ncase}(n,b,\lambda x.c(x,g(x))))
\end{array}$$

DEFINITIONS

$$\mathsf{zero}:\mathsf{Nat} \qquad \frac{n:\mathsf{Nat}}{\mathsf{succ}(n):\mathsf{Nat}}$$

$$\frac{n:\mathsf{Nat} \qquad b:B(\mathsf{zero}) \qquad \begin{array}{c}\left[\ x:\mathsf{Nat}\ \right]_x \\ c(x):C(\mathsf{succ}(x))\end{array}}{\mathsf{ncase}(n,b,c):B(n)}$$

$$\begin{array}{c}\mathsf{ncase}(\mathsf{zero},b,c)=b \\ \mathsf{ncase}(\mathsf{succ}(n),b,c)=c(n)\end{array} \qquad \frac{n=n' \qquad b=b' \qquad \overline{c(x)=c'(x)}^{\,x}}{\mathsf{ncase}(n,b,c)=\mathsf{ncase}(n',b',c')}$$

$$\frac{\mathsf{zero}=\mathsf{succ}(n)}{P} \qquad \frac{\mathsf{succ}(n)=\mathsf{succ}(n')}{n=n'}$$

$$\frac{n:\mathsf{Nat} \qquad b:B(\mathsf{zero}) \qquad \begin{array}{c}\left[\ x:\mathsf{Nat}\ ;\ \ u:B(x)\ \right]_{x,u} \\ c(x,u):B(\mathsf{succ}(x))\end{array}}{\mathsf{nrec}(n,b,c):B(n)}$$

$$\begin{array}{c}\mathsf{nrec}(\mathsf{zero},b,c)=b \\ \mathsf{nrec}(\mathsf{succ}(n),b,c)=c(n,\mathsf{nrec}(n,b,c))\end{array} \qquad \frac{n=n' \qquad b=b' \qquad \overline{c(x,u)=c'(x,u)}^{\,x,u}}{\mathsf{nrec}(n,b,c)=\mathsf{nrec}(n',b',c')}$$

$$\frac{n:\mathsf{Nat} \qquad P(\mathsf{zero}) \qquad \begin{array}{c}\left[\ x:\mathsf{Nat}\ ;\ \ P(x)\ \right]_x \\ P(\mathsf{succ}(x))\end{array}}{P(n)} \qquad n \prec_{\mathsf{pR}} \mathsf{succ}(n)$$

DERIVEDRULES

Figure 4.4: *An Encoding of the Natural Number Data Type*

$$
\begin{aligned}
\mathsf{List}(A) &\equiv \mu\, X\,.\,\mathsf{Unit} + A \times X\\
[] &\equiv \mathsf{inl}(\mathsf{one})\\
h \bullet t &\equiv \mathsf{inr}(\langle h, t\rangle)\\
\mathsf{lcase}(l, b, c) &\equiv \mathsf{when}(l, \lambda x.b, \lambda y.\mathsf{split}(y, c))\\
\mathsf{lrec}(l, b, c) &\equiv \mathsf{rec}(l, \lambda l\ g.\mathsf{lcase}(l, b, \lambda h\ t.c(h, t, g(t))))
\end{aligned}
$$

Definitions

$$
[] : \mathsf{List}(A) \qquad \frac{h : A \qquad t : \mathsf{List}(A)}{h \bullet t : \mathsf{List}(A)}
$$

$$
\frac{l : \mathsf{List}(A) \qquad b : B([]) \qquad \begin{matrix}\left[\ x : A\ ;\ \ y : \mathsf{List}(A)\ \right]_{x,y}\\ c(x, y) : B(x \bullet y)\end{matrix}}{\mathsf{lcase}(l, b, c) : B(l)}
$$

$$
\begin{matrix}\mathsf{lcase}([], b, c) = b\\ \mathsf{lcase}(h \bullet t, b, c) = c(h, t)\end{matrix} \qquad \frac{l = l' \qquad b = b' \qquad \overline{c(x, y) = c'(x, y)}^{\,x,y}}{\mathsf{lcase}(l, b, c) = \mathsf{lcase}(l', b', c')}
$$

$$
\frac{[] = h \bullet t}{P} \qquad \frac{h \bullet t = h' \bullet t' \qquad \begin{matrix}\left[\ h = h'\ ;\ \ t = t'\ \right]\\ P\end{matrix}}{P}
$$

$$
\frac{l : \mathsf{List}(A) \qquad b : B([]) \qquad \begin{matrix}\left[\ x : A\ ;\ \ y : \mathsf{List}(A)\ ;\ \ u : B(t)\ \right]_{x,y,u}\\ c(x, y, u) : B(x \bullet y)\end{matrix}}{\mathsf{lrec}(l, b, c) : B(l)}
$$

$$
\begin{matrix}\mathsf{lrec}([], b, c) = b\\ \mathsf{lrec}(h \bullet t, b, c) = c(h, t, \mathsf{lrec}(t, b, c))\end{matrix} \qquad \frac{l = l' \qquad b = b' \qquad \overline{c(x, y, u) = c'(x, y, u)}^{\,x,y,u}}{\mathsf{lrec}(l, b, c) = \mathsf{lrec}(l', b', c')}
$$

$$
\frac{l : \mathsf{List}(A) \qquad P([]) \qquad \begin{matrix}\left[\ x : A\ ;\ \ y : \mathsf{List}(A)\ ;\ \ P(y)\ \right]_{x,y}\\ P(x \bullet y)\end{matrix}}{P(l)} \qquad t \prec_{\mathsf{pR}} h \bullet t
$$

DerivedRules

Figure 4.5: *An Encoding of the List Data Type*

Boolean Valued Functions

Predicates in CCL are meta-functions (e.g. $< :: \iota \Rightarrow \iota \Rightarrow o$), whereas conditionals in the programming language $\mathcal{L}$ are boolean valued functions (e.g. $lt : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Bool}$). But there is a simple relationship between predicates and corresponding boolean valued functions. If a unary predicate P is computable, then there is a corresponding boolean valued function p such that for all well-typed a

$$P(a) \quad \leftrightarrow \quad p \,\text{`}\, a = \mathsf{true}$$

For the binary example above, for all $m, n : \mathsf{Nat}$, it can be proved that

$$m < n \;\leftrightarrow\; lt \,\text{`}\, m \,\text{`}\, n = \mathsf{true}$$

Type checking the goal $\mathsf{cond}(p \,\text{`}\, a, t, u) : A$ generates the subgoals

$$\begin{array}{r}p \,\text{`}\, a = \mathsf{true} \implies t : A \\ p \,\text{`}\, a = \mathsf{false} \implies u : A\end{array}$$

for which the equivalence above can be used to replace the hypotheses with $P(x)$ and $\neg P(x)$ respectively. These replacements are incorporated into type checking using the following derived rule

$$\frac{b : \{x{:}\mathsf{Bool}, P \;\leftrightarrow\; x = \mathsf{true}\} \qquad \begin{array}{c}\left[\, P \,\right] \\ t : A\end{array} \qquad \begin{array}{c}\left[\, \neg P \,\right] \\ u : A\end{array}}{\mathsf{cond}(b, t, u) \;:\; A}$$

and a list of rules for boolean valued functions, whose conclusions resolve with the first premise.

Tactics for Type Checking

The following tactics are used to type check goals.

- `typechk_step_tac rls n` either solves goal `n` by appeal to an assumption or uses the first applicable rule from `rls` and those rules described above;
- `typechk_tac rls` repeatedly applies `typechk_step_tac rls n` to all goals of the form $a{:}A$ in which a is at least partially instantiated (i.e. a is not a logical variable);
- `itypechk_tac s rls n` applied to a goal $?x : A$, instantiates $?x$ with the term represented by the string `s` and calls `typechk_tac rls` restricted to goal `n`.

The tactic `typechk_tac` generates correctness conditions for goals of the form $a{:}A$. Only if program a is badly typed (i.e. a is instantiated but no rule is found to reduce the goal $a{:}A$) does `typechk_tac` fail. The tactic `itypechk_tac` allows programs to be instantiated in stages during proof and type checked only as far as the instantiation. In an interactive proof, the current proof state can be taken into account when considering further instantiations.

For simple types, type checking raises conditions for termination only. For example, the program lt determines if one natural number is less than another.

$$\begin{array}{lll} lt & \equiv & \mathsf{lam}\ m.\mathsf{lam}\ n.\ \mathsf{letrec}\ \ lt\ m\ n\ \mathsf{be}\ \ \mathsf{ncase}(n, \mathsf{false}, \lambda y.\mathsf{ncase}(m, \mathsf{true}, \lambda x.\, lt\,\text{`}\,x\,\text{`}\,y)) \\ & & \qquad\qquad\qquad \mathsf{in}\ \ lt\,\text{`}\,m\,\text{`}\,n\ \ \mathsf{end} \end{array}$$

Applying `typechk_tac` to the goal

1. $lt \ :\ \ \mathsf{Nat} \rightarrow \mathsf{Nat} \rightarrow \mathsf{Nat}$

which asserts that lt is a total function over the natural numbers, generates a single condition for termination.

1. $x : \mathsf{Nat} \ \implies\ x \prec_{?R}\ \mathsf{succ}(x)$

This is solved by choosing a well-founded relation for $?R$ in which x is 'less-than' $\mathsf{succ}(x)$. In fact, the ordering pR is sufficient (see §4.4.6).

If predicates are included in types (using subtypes), then type checking generates instances of these predicates. For example, applying `typechk_tac` to the goal

1. $lt \ :\ \ \Pi m{:}\mathsf{Nat}.\Pi n{:}\mathsf{Nat}.\{x{:}\mathsf{Bool}, m < n \leftrightarrow x = \mathsf{true}\}$

reduces it to the following correctness conditions.

1. $[\![m : \mathsf{Nat};\ \ n : \mathsf{Nat}\ ]\!] \ \implies\ m < \mathsf{zero}\ \leftrightarrow\ \mathsf{false} = \mathsf{true}$

2. $[\![m : \mathsf{Nat};\ \ n : \mathsf{Nat}\ ]\!] \ \implies\ \mathsf{zero} < \mathsf{succ}(n)\ \leftrightarrow\ \mathsf{true} = \mathsf{true}$

3. $[\![m : \mathsf{Nat};\ \ n : \mathsf{Nat};\ \ m < n\ \leftrightarrow\ lt\,\text{`}\,m\,\text{`}\,n = \mathsf{true}\ ]\!] \ \implies$
   $\mathsf{succ}(m) < \mathsf{succ}(n)\ \leftrightarrow\ lt\,\text{`}\,m\,\text{`}\,n = \mathsf{true}$

4. $[\![m : \mathsf{Nat};\ \ n : \mathsf{Nat}\ ]\!] \ \implies\ \langle m, n\rangle \prec_{?R}\ \langle \mathsf{succ}(m), \mathsf{succ}(n)\rangle$

4.4.2 Rewriting

The congruence rules for bi-implication (`iff_cong_rls`) and for term and type equality (`eq_cong_rls`) allow arbitrary subformulae and subtypes to be manipulated within formulae. The Isabelle simplifier supports this style of reasoning, allowing subterms and subtypes to be rewritten using conversion rules, and subformulae to be simplified using logical equivalences.

However, the conversion rule for rec

$$\mathsf{rec}(a, h) \quad = \quad h(a, \lambda x.\mathsf{rec}(x, h))$$

can be repeatedly applied, thereby compromising termination of the simplifier. Two forms of rewriting are, therefore, considered: the Isabelle simplifier with the conversion rule for rec added only when required (with the risk of non-termination), and a purpose built rewriter which reduces all terms that appear in a goal to their canonical forms. For the Isabelle simplifier, a basic simplification set is defined containing the standard simplification set of `FOL` together with the conversion and congruence rules of `CCL`.

The following tactics are used for rewriting.

- `asm_simp_tac rs cs n` uses `ASM_SIMP_TAC` from the simplifier with the additional rewrite rules `rs` and congruence rules `cs` to simplify goal `n`;

- `asm_simp_case_tac rs cs n` uses `ASM_SIMP_CASE_TAC` from the simplifier with the additional rewrite rules `rs` and congruence rules `cs` to simplify goal `n`;

- `eval_tac rls n` solves goal `n` of the form $t = a$, where a is the canonical form of t, by depth-first search using derived evaluation rules for $\mathcal{L}$;
- `equal_tac rls` reduces every term in a goal to canonical form by breaking up formulae using the congruence rules for predicates (`iff_cong_rls`) and then applying `eval_tac` to goals of the form $t = ?x$.

The tactics `asm_simp_tac` and `asm_simp_case_tac` are the most commonly used for rewriting. The tactic `eval_tac` is an interpreter for $\mathcal{L}$; a program t is evaluated by solving the goal $t = ?x$. For example, if $\underline{0}$ represents zero, $\underline{1}$ represents $\mathsf{succ}(\mathsf{zero})$ etc., then an application of lt may be evaluated by solving the goal

1. $lt \text{ ‘ } \underline{2} \text{ ‘ } \underline{6} \; = ?x$

using `eval_tac []`. This succeeds and $?x$ becomes instantiated with the canonical form of $lt \text{ ‘ } \underline{2} \text{ ‘ } \underline{6}$.

$$?x \;\longleftarrow\; \mathsf{true}$$

4.4.3 Introducing Recursive Functions

A program specification typically begins with a series of Π type-formers prescribing the argument types.

$$?prog \;:\; \Pi x_1{:}A_1 \ldots \Pi x_n{:}A_n . B(x_1, \ldots\ x_n)$$

in which each A_j may depend on all x_i for $i < j$. One program satisfying this specification has the form

$$\mathsf{lam}\ x_1 \ldots \mathsf{lam}\ x_n . \mathsf{letrec}\ f\, x_1 \ldots x_n\ \mathsf{be}\ ?a(x_1 \ldots x_n, f)\ \mathsf{in}\ f \text{ ‘ } x_1 \ldots \text{ ‘ } x_n\ \mathsf{end}$$

The tactic `rec_tac` instantiates a program with this form by repeatedly applying the rule lam-*type* and then applying letrec-*type* with the appropriate number of arguments. The initial specification is replaced by the subgoal

1. $[\![\; x_1 : A_1, \ldots x_n : A_n;$
   $\forall y_1{:}A_1 \ldots \forall y_n{:}A_n . \langle y_1, \ldots y_n\rangle \;\prec_{?R}\; \langle x_1, \ldots x_n\rangle \supset f \text{ ‘ } y_1 \ldots \text{ ‘ } y_n : B(y_1, \ldots y_n) \;]\!] \implies$
   $?a(x_1, \ldots x_n, f) : B(x_1, \ldots x_n)$

leaving the body of the program $?a$ to be synthesised for well-typed arguments $x_1 \ldots x_n$, assuming an induction hypothesis for recursive calls to f. For example, a division algorithm may be specified as

1. $?divide \;:\; \Pi n{:}\mathsf{Nat}.\Pi d{:}\{d{:}\mathsf{Nat}, \neg d = \mathsf{zero}\}.\{q{:}\mathsf{Nat}, DIV(n, d, q)\}$

where $DIV(n, d, q)$ holds iff q is the integer quotient of n divided by d. Applying `rec_tac` instantiates $?divide$ with a template for a recursive program.

$$?divide \longleftarrow \mathsf{lam}\ n.\mathsf{lam}\ d.\mathsf{letrec}\ div\, n\ d\ \mathsf{be}\ ?a(n, d, div)\ \mathsf{in}\ div \text{ ‘ } n \text{ ‘ } d\ \mathsf{end}$$

and leaves the goal

1. $[\![\; n : \mathsf{Nat};\;\; d : \{x{:}\mathsf{Nat}, \neg x = \mathsf{zero}\};$
   $\forall u{:}\mathsf{Nat}.\forall v{:}\{x{:}\mathsf{Nat}, \neg x = \mathsf{zero}\}.$
   $\langle u, v\rangle \;\prec_{?R}\; \langle n, d\rangle \supset div \text{ ‘ } u \text{ ‘ } v : \{q{:}\mathsf{Nat}, DIV(u, v, q)\} \;]\!] \implies$
   $?a(n, d, div) : \{q{:}\mathsf{Nat}, DIV(n, d, q)\}$

The induction hypothesis allows the behaviour of any recursive call $div \text{‘} x \text{‘} y$ to be assumed provided that the pair $\langle x, y\rangle$ is less-than the pair $\langle n, d\rangle$ in some, as yet undetermined, well-founded ordering $?R$.

4.4.4 Introducing Local Declarations

The type rules for let, letfun and letrec introduce uninstantiated type variables. For example, applying let-*type*

$$\frac{a : A \qquad \begin{matrix}\left[\; x : A \;\right]_x \\ b(x) : B\end{matrix}}{\mathsf{let}\ x\ \mathsf{be}\ a\ \mathsf{in}\ b(x)\ \mathsf{end} : B}$$

introduces a new type $?A$. Proving $a{:}?A$ first may instantiate $?A$ with a type that is too general for the second premise to hold; proving $x{:}?A \Longrightarrow b(x) : B$ first may instantiate $?A$ with a type that a does not inhabit. As they introduce uninstantiated type variables, these rules are too undirected to be used in a general tactic for type checking, except for simple instances of letfun-*type* and letrec-*type* in which the body is a direct application of the declared function (see, for example, the definition of *lt* in §4.4.1).

Instead, tactics are provided that explicitly introduce local declarations by hand. For example, the type rules for letfun introduce uninstantiated specifications for the auxiliary functions. The tactic `letfun_tac ls n` takes a list of type instantiations `ls`, of the form $[A_1, \ldots A_n, \lambda x_1 \ldots x_n B(x_1 \ldots x_n)]$, and a goal `n`. It applies a letfun-*type* rule to goal `n`, with the auxiliary types instantiated by the corresponding elements of `ls`. The appropriate goal is inferred from the length of `ls`.

In the same way, recursive auxiliary functions, declared using letrec, are introduced with the tactic `letrec_tac`. For example, in the development of an algorithm for insertion sort, the following goal arises.

1. $[\![\ h : A;\ \ t : \mathsf{List}(A)\ ]\!] \implies ?a(h,t) : \{x{:}\mathsf{List}(A), \mathit{SORT}(h \bullet t, x)\}$

where $\mathit{SORT}(l,m)$ holds iff m is an ordered permutation of l. An auxiliary function for insertion is specified by the tactic

```
letrec_tac [A,{l:List(A),ORD(l)}, %a l.{x:List(A),SORT(a::l,x)}] 1
```

where $\mathit{ORD}(l)$ holds iff l is ordered. This instantiates

$$?a(h,t) \longleftarrow \mathsf{letrec}\ \mathit{insert}\, x\ l\ \mathsf{be}\ ?a(x,l)\ \mathsf{in}\ ?b(h,t,\mathit{insert})\ \mathsf{end}$$

and produces the goals

1. $\forall u{:}A.\forall v{:}\{x{:}\mathsf{List}(A), \mathit{ORD}(x)\}.\mathit{insert}\text{ ‘ }u\text{ ‘ }v : \{x{:}\mathsf{List}(A), \mathit{SORT}(u \bullet v, x)\} \implies ?b(h,t,\mathit{insert}) : \{x{:}\mathsf{List}(A), \mathit{SORT}(h \bullet t, x)\}$
2. $[\![\ u : A;\ \ v : \{x{:}\mathsf{List}(A), \mathit{ORD}(x)\}\ ]\!] \implies ?a(u,v) : \{x{:}\mathsf{List}(A), \mathit{SORT}(u \bullet v, x)\}$

The body of the sorting algorithm is then derived by solving the first subgoal, assuming the existence of an *insert* function; the second subgoal specifies this function.

4.4.5 Properties of Recursive Calls

The induction hypothesis produced by rec-*type* (or letrec-*type*) asserts the type of recursive calls, including any properties that this may imply. In the example for *lt* in §4.4.1, the induction hypothesis was instantiated by type checking. But if recursive calls occur inside constructors, then this does not happen (by modifying the type rules this can be avoided, see §6.2.2). Instead, the following derived rule generates the appropriate instantiations of the induction hypothesis for each correctness condition.

$$\frac{\forall y{:}A.y \prec_R x \supset g(y) : \{z{:}B, P(z)\} \qquad \begin{array}{c}\left[\; g(a) : B \;;\; P(g(a)) \;\right] \\ Q(g(a))\end{array} \qquad a : A \qquad a \prec_R x}{Q(g(a))}$$

Resolution with this rule uses unification to specialise an induction hypothesis produced by rec-*type* to an instance of the recursive call present in the current subgoal.

Similar rules are derived for letrec. For example, the following goal specifies a complete division algorithm.

1. $\mathsf{lam}\ n.\mathsf{lam}\ d.\mathsf{letrec}\ \ div\ n\ d\ \ \mathsf{be}\ \ \mathsf{cond}(lt\ `\ n\ `\ d, \mathsf{zero}, \mathsf{succ}(div\ `\ (n-d)\ `\ d))$
   $\mathsf{in}\ \ div\ `\ n\ `\ d\ \ \mathsf{end}\ \ :\ \Pi n{:}\mathsf{Nat}.\Pi d{:}\{x{:}\mathsf{Nat}, \neg x = \mathsf{zero}\}.\{q{:}\mathsf{Nat}, DIV(n,d,q)\}$

Type checking raises a set of correctness conditions, including the following one for the recursive call.

1. $[\![\ n : \mathsf{Nat};\ \ d : \{x{:}\mathsf{Nat}, \neg x = \mathsf{zero}\};$
   $\forall u{:}\mathsf{Nat}.\forall v{:}\{x{:}\mathsf{Nat}, \neg x = \mathsf{zero}\}.$
   $\langle u, v\rangle \prec_{?R} \langle n, d\rangle \supset div\ `\ u\ `\ v : \{q{:}\mathsf{Nat}, DIV(u,v,q)\}\ ]\!]\ \Longrightarrow$
   $\mathsf{succ}(div\ `\ (n-d)\ `\ d) : \{q{:}\mathsf{Nat}, DIV(n,d,q)\}$

Applying the derived rule for letrec specialises the induction hypothesis to give, after some type checking,

1. $[\![\ n : \mathsf{Nat};\ \ d : \{x{:}\mathsf{Nat}, \neg x = \mathsf{zero}\};\ \ DIV(n-d, d, div\ `\ (n-d)\ `\ d)\ ]\!]\ \Longrightarrow$
   $DIV(n, d, \mathsf{succ}(div\ `\ (n-d)\ `\ d))$

2. $[\![\ n : \mathsf{Nat};\ \ d : \{x{:}\mathsf{Nat}, \neg x = \mathsf{zero}\}\ ]\!]\ \Longrightarrow\ \langle n-d, d\rangle \prec_{?R} \langle n, d\rangle$

The first goal is the correctness condition with an instance of the induction hypothesis for the recursive call $div\ `\ (n-d)\ `\ d$. The second goal is the termination condition for this recursive call.

4.4.6 Well-Founded Orderings

When a program is type checked, each recursive call gives rise to a subgoal of the form

$$a \prec_R b$$

These are solved using the rules for well-founded orderings (Figure 3.4).

The tactic `prel_tac` solves a restricted class of these goals whether or not R is instantiated. `prel_tac` succeeds if R is, or can be instantiated to, the primitive ordering over terms (pR) or a projection from tuples to the primitive ordering and the goal can be solved without using the transitivity of pR, otherwise the tactic fails—it always terminates.

If a and b are tuples, then `prel_tac` considers each element of the tuples in turn using the following projections.

$$\begin{aligned}\mathsf{wfst}(R) &\equiv \mathsf{map}(\lambda p.\mathsf{split}(p, \lambda x\ y.x), R)\\ \mathsf{wsnd}(R) &\equiv \mathsf{map}(\lambda p.\mathsf{split}(p, \lambda x\ y.y), R)\end{aligned}$$

For example, the following are solved by `prel_tac`

$$\begin{aligned} \langle n, a\rangle &\prec_{?R} \langle \mathsf{succ}(n), a'\rangle \\ \langle a, \langle b, t\rangle\rangle &\prec_{?R} \langle a', \langle b', h \bullet t\rangle\rangle \end{aligned}$$

instantiating $?R$ with $\mathsf{wfst}(\mathsf{pR})$ and $\mathsf{wsnd}(\mathsf{wsnd}(\mathsf{pR}))$ respectively.

The set of functions for which `prel_tac` succeeds includes all those that are primitive recursive in one of their arguments.

4.5 Interpretation of First-Order Theories

Dependent types and subtypes are convenient for specifying programs and stating their correctness with respect to these specifications. But proving simple facts about particular data (e.g. that addition on natural numbers is commutative) is more convenient in simply typed logics in which type checking is decidable and so can be relegated to a well-formedness condition (e.g. typed first-order logic and higher-order logic).

To define new data types easily and state program correctness clearly, CCL uses the generalised type-formers Π and Σ, and subtypes. But CCL admits a direct interpretation of first-order logic extended with computational data types, which provides formal justification for the extensions and allows theorems proved in the first-order logic to be lifted up to theorems of CCL. In this sense, CCL acts as a meta-theory in which to formalise first-order logic with a collection of computational types. Not every term-former of CCL has a simply typed counterpart; the constant rec cannot be simply typed, although primitive recursive functionals such as nrec can.

As Isabelle does not yet support theory interpretation, the interpretation of first-order logic with computational types was carried out by hand. But the theorems in CCL corresponding to axioms of first-order logic were all derived using Isabelle, so that the reasoning done by hand was straightforward and kept to a minimum.

This section describes how interpretations can be set up for simple data types, for sorts of data types that admit certain properties and for abstract data types in which equality is interpreted by a congruence relation other than term equality. These are all based on Isabelle's implementation of classical first-order logic `FOL`, described in §4.1.2.

4.5.1 Simple Data Types

`FOL` can be extended with a meta-type of booleans *bool* (giving `FOL+BOOL`). The extension `BOOL` has constants,

$$\begin{aligned} \mathsf{true} &:: \mathit{bool} \\ \mathsf{false} &:: \mathit{bool} \\ \mathsf{cond} &:: \mathit{bool} \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \alpha \end{aligned}$$

and axioms for conversion, the freeness of constructors and induction (case analysis).

$$\begin{aligned} \mathsf{cond}(\mathsf{true}, c, d) &= c \\ \mathsf{cond}(\mathsf{false}, c, d) &= d \end{aligned} \qquad \neg\mathsf{true} = \mathsf{false} \qquad \frac{P(\mathsf{true}) \qquad P(\mathsf{false})}{P(b)}$$

where $P :: \mathit{bool} \Rightarrow o$, restricting its application to terms of meta-type *bool*.

By mapping meta-type membership (::) on to type membership in CCL (:) and $= :: \mathit{bool} \Rightarrow \mathit{bool} \Rightarrow o$ on to term equality, this extension is directly interpreted by the type

Bool. The meta-types of the constants are simply typed versions of the type rules for their namesakes in CCL. Conversion, freeness and induction all follow from the theorems for Bool in Figure 4.2.

Similarly, a meta-type of natural numbers (nat) can be added (giving `FOL+BOOL+NAT`). The extension `NAT` has constants

$$\begin{aligned}\mathsf{zero} &:: nat\\ \mathsf{succ} &:: nat \Rightarrow nat\\ \mathsf{ncase} &:: nat \Rightarrow (nat \Rightarrow \alpha) \Rightarrow \alpha\\ \mathsf{nrec} &:: nat \Rightarrow (nat \Rightarrow \alpha \Rightarrow \alpha) \Rightarrow \alpha\end{aligned}$$

and axioms for conversion, congruence (of ncase and nrec), the freeness of constructors and induction. Again, there is a direct interpretation justifying the extension, which maps the meta-type nat to the type Nat and $= :: nat \Rightarrow nat \Rightarrow o$ to term equality.

4.5.2 Sorts of Data Types

In the programming language Haskell, *types* can belong to *classes* [29]. Membership of a class implies that a type supports certain operations. Similarly in Isabelle, meta-types belong to sorts. Type variables in axioms can be restricted to range over the types of a particular sort, so that membership of a sort implies that a type has certain (axiomatised) properties.

Computational types in extended first-order logic can belong to sorts in the same way as types in Haskell belong to classes. The sorts of computational types are interpreted in CCL as predicates over types (since there is a universe of types, τ). These predicates formalise the properties of sorts. Inclusion of a type in some sort a is justified by proving that the interpretation of the type in CCL satisfies the predicate which interprets a. For example, some data types in CCL admit a test for equality. Consider the definition

$$EQTYPE(A) \equiv \Pi a{:}A.\Pi b{:}A.\{x{:}\mathsf{Bool}, x = \mathsf{true} \leftrightarrow a = b\}$$

to which the sort $eqterm \Subset term$ in extended first-order logic corresponds. For any type A in CCL, if the following holds for some program eq_A

$$\mathsf{eq}_A : EQTYPE(A)$$

then a meta-type α interpreted by A is included in the sort $eqterm$. Isabelle can automatically infer sort membership from properties of type-formers. In this case, the product type-former preserves membership of $eqterm$ whereas the function type-former does not.

4.5.3 Abstract Data Types

The substitution rule in `FOL` implies that equality on each meta-type is a congruence relation. Any interpretation of equality must reflect this. In the extensions to `FOL` described above, equality on each new meta-type A ($= :: A \Rightarrow A \Rightarrow o$) is mapped to term equality in CCL ($= :: \iota \Rightarrow \iota \Rightarrow o$), so that any implied congruence axioms will trivially hold in the interpretation.

Abstract data types are encoded in much the same way as they are in programming languages. A representation type is defined in which the abstract type and operations defined over it are implemented. Equality between objects of this abstract type is then interpreted in CCL by some relation weaker than term equality. But in the interpretation of an abstract type, the implied congruence axioms must be proved for the relation interpreting equality. This is illustrated by the following development of a theory of finite sets, in which equality between sets is defined to reflect the idempotent and commutative nature of set 'cons'.

$\alpha_=$ *set* is the meta-type of finite sets of elements of equality type $\alpha_=$. SET introduces the meta-type former *set* and constants for the empty set $\varnothing$, set 'cons' $\odot$, set membership mem, subsets subseteq, cardinality card, collection $\{\cdot \mid \cdot\}$, replacement $\{\cdot \mid \cdot\ ,\ \cdot\}$, union $\bigcup$ and power set Pow. Write finite sets $x \odot \varnothing$, $x \odot y \odot \varnothing$ etc. as $\{\, x \,\}$, $\{\, x, y \,\}$ etc. The axioms of SET (Figure 4.6) provide

- induction over sets,
- conversions for mem, subseteq and card,
- set axioms for extensionality, collection, replacement, union and power sets, and
- congruence rules for the higher-order constants $\{\cdot \mid \cdot\}$ and $\{\cdot \mid \cdot\ ,\ \cdot\}$.

Predicates for membership and subsets, and functions for set union, intersection and difference are defined (Figure 4.6). The usual theorems for finite sets can be developed within SET.

SET is interpreted in CCL by mapping $\alpha_=$ *set* on to the representation type $\mathsf{List}(T)$, where T is the equality type that interprets $\alpha_=$, and defining a new relation $=_{set} :: \iota \Rightarrow \iota \Rightarrow o$ for set equality. The relation $=_{set}$ is defined in CCL (using mem, $\in$ and $\subseteq$) as follows:

$$\begin{aligned}
a \ \mathsf{mem}\ A &\equiv \mathsf{lrec}(A, \mathsf{false}, \lambda x\ X\ g.(a\ \mathsf{eq}_T\ x)\ \mathsf{or}\ g) \\
a \in A &\equiv a\ \mathsf{mem}\ A = \mathsf{true} \\
A \subseteq B &\equiv \forall x.\ x \in A \supset x \in B \\
A =_{set} B &\equiv A \subseteq B \wedge B \subseteq A
\end{aligned}$$

It is simple to show that $=_{set}$ is an equivalence relation. As intended, list 'cons' ($\bullet$) is idempotent and commutative with respect to $=_{set}$—the following are theorems in CCL.

$$\begin{aligned}
x \bullet x \bullet A &=_{set} x \bullet A \\
x \bullet y \bullet A &=_{set} y \bullet x \bullet A
\end{aligned}$$

The constants of SET are mapped on to the following terms in CCL.

$$\begin{aligned}
\varnothing &\mapsto [] \\
\odot &\mapsto \bullet \\
a\ \mathsf{mem}\ A &\mapsto \mathsf{lrec}(A, \mathsf{false}, \lambda x\ X\ g.(a\ \mathsf{eq}_T\ x)\ \ \mathsf{or}\ \ g) \\
A\ \mathsf{subseteq}\ B &\mapsto \mathsf{lrec}(A, \mathsf{true}, \lambda x\ X\ g.(x\ \mathsf{mem}\ B)\ \ \mathsf{and}\ \ g) \\
\mathsf{card}(A) &\mapsto \mathsf{lrec}(A, [], \lambda x\ X\ g.\mathsf{cond}(x\ \mathsf{mem}\ X, g, \mathsf{succ}(g))) \\
\{x \in A \mid p(x)\} &\mapsto \mathsf{lrec}(A, [], \lambda x\ X\ g.\mathsf{cond}(p(x), x \bullet g, g)) \\
\{y \mid x \in A\ ,\ y = f(x)\} &\mapsto \mathsf{lrec}(A, [], \lambda x\ X\ g.f(x) \bullet g) \\
A \cup B &\mapsto \mathsf{lrec}(A, B, \lambda x\ X\ g.x \bullet g) \\
\bigcup(A) &\mapsto \mathsf{lrec}(A, [], \lambda x\ X\ g.x \cup g) \\
\mathsf{Pow}(A) &\mapsto \mathsf{lrec}(A, [] \bullet [], \lambda x\ X\ g.\{y \mid z \in g\ ,\ y = x \bullet z\}\ \cup\ g)
\end{aligned}$$

Note that set collection is mapped on to the list functional *filter*, replacement on to *map*, $\cup$ on to the function *append* and $\bigcup$ on to a function that flattens lists of lists down to

$$\neg\varnothing = a \odot A \qquad \frac{P(\varnothing) \qquad \begin{array}{c}\left[\ P(X)\ \right]_{x,X} \\ P(x \odot X)\end{array}}{P(A)}$$

$$\begin{array}{rr} a \ \mathsf{mem}\ \varnothing = \mathsf{false} & a \ \mathsf{mem}\ b \odot A = (a \ \mathsf{eq}_T\ b) \ \mathsf{or}\ a \ \mathsf{mem}\ A \\ \varnothing \ \mathsf{subseteq}\ A = \mathsf{true} & a \odot A \ \mathsf{subseteq}\ B = (a \ \mathsf{mem}\ B) \ \mathsf{and}\ (A \ \mathsf{subseteq}\ B) \\ \mathsf{card}(\varnothing) = \mathsf{zero} & \mathsf{card}(a \odot A) = \mathsf{cond}(a \ \mathsf{mem}\ A, \mathsf{card}(A), \mathsf{succ}(\mathsf{card}(A))) \end{array}$$

$$\begin{array}{rcl} A = B & \leftrightarrow & (\forall\, x.\ x \in A \leftrightarrow x \in B) \\ a \in \{x \in A \mid p(x)\} & \leftrightarrow & a \in A \wedge p(a) = \mathsf{true} \\ a \in \{y \mid x \in A\ ,\ y = f(x)\} & \leftrightarrow & (\exists\, y.\ y \in A \wedge a = f(y)) \\ a \in \bigcup A & \leftrightarrow & (\exists\, X.\ X \in A \wedge a \in X) \\ a \in \mathsf{Pow}(A) & \leftrightarrow & a \subseteq A \end{array}$$

$$\frac{A = A' \qquad \overline{p(x) = p'(x)}^{\,x}}{\{x \in A \mid p(x)\} = \{x \in A' \mid p'(x)\}}$$

$$\frac{A = A' \qquad \overline{f(x) = f'(x)}^{\,x}}{\{y \mid x \in A\ ,\ y = f(x)\} = \{y \mid x \in A'\ ,\ y = f'(x)\}}$$

Axioms

$$\begin{array}{rcl} a \in A & \equiv & a \ \mathsf{mem}\ A = \mathsf{true} \\ A \subseteq B & \equiv & A \ \mathsf{subseteq}\ B = \mathsf{true} \\ A \subset B & \equiv & A \subseteq B \ \wedge\ \neg A = B \\ A \cup B & \equiv & \bigcup\{\, A, B\,\} \\ A \cap B & \equiv & \{x \in A \mid x \in B\} \\ A - B & \equiv & \{x \in A \mid \neg x \in B\} \end{array}$$

Definitions

Figure 4.6: *Axioms for Finite Set Theory*

lists. Under this interpretation the axioms of Figure 4.6 are theorems in CCL. In addition, the following congruences hold (implied by the properties of equality in `FOL`), which justify the interpretation of $= :: set \Rightarrow set \Rightarrow o$ by $=_{set} :: \iota \Rightarrow \iota \Rightarrow o$.

$$\frac{x = x' \qquad A =_{set} A'}{x \bullet A =_{set} x' \bullet A'} \qquad \frac{x = x' \qquad A =_{set} A'}{x \ \mathsf{mem}\ A = x' \ \mathsf{mem}\ A'}$$

$$\frac{A =_{set} A' \qquad B =_{set} B'}{A \subseteq B \leftrightarrow A' \subseteq B'} \qquad \frac{A =_{set} A'}{\mathsf{card}(A) = \mathsf{card}(A')}$$

$$\frac{A =_{set} A' \qquad \overline{p(x) = p'(x)}^{\,x}}{\{x \in A \mid p(x)\} =_{set} \{x \in A' \mid p'(x)\}}$$

$$\frac{A =_{set} A' \qquad \overline{f(x) = f'(x)}^{\,x}}{\{y \mid x \in A\ ,\ y = f(x)\} =_{set} \{y \mid x \in A'\ ,\ y = f'(x)\}}$$

$$\frac{A =_{set} A'}{\bigcup A =_{set} \bigcup A'} \qquad \frac{A =_{set} A'}{\mathsf{Pow}(A) =_{set} \mathsf{Pow}(A')}$$

4.6 Summary of Implementation

There are, effectively, two separate implementations: the theory CCL and first-order logic with extensions for computational types. Translation between these two theories is by hand; interpreting the axiomatisation of new types in `FOL` by types in `CCL`, and lifting lemmas proved in an extension of `FOL` up to theorems in `CCL`. But these translations are straightforward, requiring little more than cutting and pasting formulae between theories. For each new meta-type added to `FOL`, theorems are proved in `CCL` that correspond to the axioms added to `FOL`.

The Isabelle theory `CCL` is an extension of the existing theory `FOL`. From the axioms described in Chapter 3, the theorems of §4.2 are derived. For defined data types, the rules described in §4.3 are proved using simple tactics. The major piece of outstanding work required to automate this process is a parser for syntax in the style of an ML `datatype` statement. The complete set of tactics used in `CCL` is a little over 100 lines of ML code. The extensions to Isabelle's `FOL` are axiomatised in §4.5. The standard Isabelle tactics are used for natural deduction inference (`fast_tac`) and rewriting and induction (using the simplifier). The fact that standard tactics already exist is one reason that these theories were used. They are also more efficient, as there is more information relevant to proof search in the meta-logic.

Chapter 5

Proving Programs Correct

This chapter describes how the implementation of CCL is used to reason about program correctness.

In CCL, both termination and correctness are expressed as goals of the form $a : A$, which allows type checking to direct proofs. Correctness proofs are done in the following three stages: correctness is stated as a goal $a : A$, type checking reduces this goal to a set of correctness conditions, and these are then translated to an appropriate first-order logic and proved. Verification and derivation differ only in the degree to which a is initially instantiated. To give an indication of how suited CCL is to reasoning about program correctness, the derivation of Robinson's unification algorithm is presented as an extended example.

§5.1 describes how CCL is used to reason about termination. §5.2 describes how specifications are developed in CCL. Verification and derivation are introduced in §5.3 and §5.4 respectively. In each case, separate goals arise for termination, which are identical to those described in §5.1. §5.5 describes how correctness conditions are proved in first-order logic. Finally, §5.6 presents the extended example: unification. Hypotheses that are redundant or just assert simple type information are sometimes omitted from goals in the following examples to maintain legibility.

5.1 Termination

If a program of $\mathcal{L}$ is typeable in CCL, then it terminates (Theorem 3.10). Primitive recursive functions can be defined using functionals such as nrec and lrec introduced in §4.3.2. Proving that these functions terminate is simply a matter of type checking—as in Martin-Löf's Type Theory, the type rules for primitive recursive functionals ensure termination. For example, the following goal asserts that an addition algorithm is total over the type Nat.

1. $\mathsf{lam}\ m.\mathsf{lam}\ n.\mathsf{nrec}(m, n, \lambda x\ g.\mathsf{succ}(g))\ :\ \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$

Applying `typechk_tac` immediately solves the goal.

More generally, recursive programs are defined using the destructor rec or the derived forms of letrec. In these cases, type checking raises a termination condition for each recursive call in the program of the form

$$x \prec_R y$$

where x is the tuple of arguments in the recursive call, and y is the tuple of formal parameters over which recursion is defined. These goals are solved and termination is proved by instantiating $?R$ with a well-founded ordering that orders every recursive call. If recursion is primitive in one of its arguments, then the tactic `prec_tac` finds a suitable ordering (namely a projection on to the primitive ordering pR) and automatically solves the termination conditions. For example, the primitive recursive algorithm for addition described above can be written using letrec and ncase instead of nrec. The following goal asserts that this version of addition is also total over the type Nat.

$$1.\ \mathsf{lam}\ m.\mathsf{lam}\ n.\ \mathsf{letrec}\ \ \mathit{add}\ m\ n\ \ \mathsf{be}\ \ \mathsf{ncase}(m, n, \lambda x.\mathsf{succ}(\mathit{add}\ `\ x\ `\ n))\ \ \mathsf{in}\ \ \mathit{add}\ `\ m\ `\ n\ \ \mathsf{end}\ :\ \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$$

Applying `typechk_tac` reduces the goal to a single condition on termination (for the single recursive call).

1. $\langle x, n\rangle \prec_{?R} \langle \mathsf{succ}(x), n\rangle$

This is primitive recursive in its first argument, and so is immediately solved by `prec_tac`. Note that there is no disadvantage in considering recursion over a tuple of the function's arguments.

In more complex recursion schemes, type checking still raises termination conditions, but these may not be solved by `prec_tac`. For example, the following goal asserts that Ackermann's function is total over the type Nat.

$$\begin{array}{lll} 1.\ \mathsf{lam}\ m.\mathsf{lam}\ n.\ \mathsf{letrec}\ \ \mathit{ack}\ m\ n\ \ \mathsf{be} & \mathsf{ncase}(m, \mathsf{succ}(n), & \\ & \quad \lambda x.\mathsf{ncase}(n,\ \mathit{ack}\ `\ x\ `\ \mathsf{succ}(\mathsf{zero}) & \\ & \qquad\qquad \lambda y.\mathit{ack}\ `\ x\ `\ (\mathit{ack}\ `\ \mathsf{succ}(x)\ `\ y))) & \\ \qquad\qquad \mathsf{in}\ \ \mathit{ack}\ `\ m\ `\ n\ \ \mathsf{end} & :\ \ \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} & \end{array}$$

Applying `typechk_tac` reduces the goal to the following three conditions on termination (one for each recursive call).

1. $\langle x, \mathsf{succ}(\mathsf{zero})\rangle \prec_{?R} \langle \mathsf{succ}(x), \mathsf{zero}\rangle$
2. $\langle \mathsf{succ}(x), y\rangle \prec_{?R} \langle \mathsf{succ}(x), \mathsf{succ}(y)\rangle$
3. $\langle x, \mathit{ack}\ `\ \mathsf{succ}(x)\ `\ y\rangle \prec_{?R} \langle \mathsf{succ}(x), \mathsf{succ}(y)\rangle$

Nested recursion is automatically handled in type checking. By the induction hypothesis generated from the rule letrec-*type*, the inner call $\mathit{ack}`\mathsf{succ}(x)`y$ is of type Nat if subgoal (2) holds. Using the type of the inner call and the induction hypothesis again, the outer call is of type Nat if subgoal (3) holds. In fact, the nested recursion in Ackermann's function is rather simple—termination depends only on the nested call being simply typed. In the unification algorithm, presented in §5.6, termination depends on deeper properties of the nested call.

The termination conditions for Ackermann's function cannot be solved by `prec_tac`. The following lexicographic ordering is needed.

$$\langle a_1, a_2\rangle \prec \langle b_1, b_2\rangle \quad \mathit{iff} \quad a_1 < b_1 \lor (a_1 = b_1 \land a_2 < b_2)$$

In CCL, the slightly weaker relation $\mathsf{lex}(\mathsf{pR}, \mathsf{pR})$ can be used. The axiom for lex (see Figure 3.4) and the derived rule $n \prec_{\mathsf{pR}} \mathsf{succ}(n)$ (see Figure 4.4) allow the termination conditions to be solved.

Termination conditions may depend on complex properties. For example, the following goal asserts that a division algorithm is total over Nat for non-zero denominators (the algorithm clearly diverges for a denominator of zero).

1. $\mathsf{lam}\ n.\mathsf{lam}\ d.\ \mathsf{letrec}\ \ \mathit{div}\ n\ d\ \ \mathsf{be}\ \ \mathsf{cond}(\mathit{lt}\ `\ n\ `\ d, \mathsf{zero}, \mathsf{succ}(\mathit{div}\ `\ (n-d)\ `\ d))$
   $\mathsf{in}\ \ \mathit{div}\ `\ n\ `\ d\ \ \mathsf{end}\ \ :\ \ \mathsf{Nat} \rightarrow \{x{:}\mathsf{Nat}, \neg x = \mathsf{zero}\} \rightarrow \mathsf{Nat}$

Applying `typechk_tac` reduces the goal to the following termination condition.

1. $[\![\ n : \mathsf{Nat};\ \ d : \{x{:}\mathsf{Nat}, \neg x = \mathsf{zero}\}\ ]\!]\ \ \Longrightarrow\ \langle n, n-d\rangle\ \prec_{?R}\ \langle n, d\rangle$

Assuming the following lemmas for all x and y in Nat, provable in extended first-order logic, the goal is solved by instantiating $?R \longleftarrow \mathsf{wsnd}(\mathsf{pR})$.

$$\begin{aligned} \neg y = \mathsf{zero}\ &\supset\ x - y < x \\ x < y\ &\supset\ x \prec_{\mathsf{pR}} y \end{aligned}$$

5.2 Specifying Correctness

In CCL, simple types provide a coarse level of specification that ensures termination. As described in §2.2, subtypes and the generalised function type-former Π can be used to specify the input/output behaviour of programs. The type

$$\Pi x{:}\{x{:}A, P(x)\}.\{y{:}B, Q(x,y)\}$$

is inhabited by functions which, when applied to any $a{:}A$ such that $P(a)$ holds, terminate with a result $b : B$ such that the input/output relation $Q(a,b)$ holds.

The connectives of first-order logic together with equality and type membership allow specifications to be conveniently expressed. For example, assuming programs for the usual arithmetic operators $+$, $\times$ and $-$, and a boolean valued function lt corresponding to the predicate $<$ (see §4.4.1), the property of being the integer quotient q of n divided by d is expressed as

$$\mathit{DIV}(n,d,q)\ \ \equiv\ \ \exists\, r{:}\mathsf{Nat}.r < d\ \wedge\ n = q \times d + r$$

Note that $\mathit{DIV}(n,d,q)$ is unsatisfiable if d equals zero, since there exists no r such that $r < \mathsf{zero}$. A program for division using non-zero divisors is specified by the following type.

$$\Pi n{:}\mathsf{Nat}.\Pi d{:}\{x{:}\mathsf{Nat}, \neg x = \mathsf{zero}\}.\{q{:}\mathsf{Nat}, \mathit{DIV}(n,d,q)\}$$

Similarly, the property of being the minimum element of a list l of type Nat is simply

$$\mathit{MINL}(l,a)\ \ \equiv\ \ a\ \mathit{in}\ l\ \wedge\ \forall\, x{:}\mathsf{List}(\mathsf{Nat}).x\ \mathit{in}\ l\ \supset\ a \leq x$$

where in is an infix predicate that holds iff the first argument is an element of the second. The predicate in is defined in terms of the program inp by

$$\begin{aligned} \mathit{inp}\ &\equiv\ \mathsf{lam}\ l.\mathsf{lam}\ a.\mathsf{lrec}(l, \mathsf{false}, \lambda x\ xs\ g.(a\ \mathsf{eq}_{nat}\ x)\ \ \mathsf{or}\ \ g) \\ a\ \mathit{in}\ l\ &\equiv\ \mathit{inp}\ `\ l\ `\ a = \mathsf{true} \end{aligned}$$

A program for finding the minimum element of a non-empty list of elements of type Nat is specified by the following type.

$$\Pi l{:}\{x{:}\mathsf{List}(\mathsf{Nat}), \neg x = []\}.\{a{:}\mathsf{Nat}, \mathit{MINL}(l,a)\}$$

A sorting algorithm should output an ordered permutation of its input. If $\leq$ is a computable ordering of some equality type A (with corresponding program le), then a predicate ORD is defined in terms of the program ord by

$$\begin{aligned} ord &\equiv \mathsf{lam}\ l.\mathsf{lrec}(l, \mathsf{true}, \lambda x\ xs\ g.\mathsf{lcase}(xs, \mathsf{true}, \lambda y\ ys.(le\ `\ x\ `\ y)\ \mathsf{and}\ g)) \\ ORD(l) &\equiv ord\ `\ l = \mathsf{true} \end{aligned}$$

so that $ORD(l)$ holds iff the list l is ordered. Permutations are defined in terms of the program $noccs$, which returns the number of occurrences of a term a in the list l.

$$\begin{aligned} noccs &\equiv \mathsf{lam}\ a.\mathsf{lam}\ l.\mathsf{lrec}(l, \mathsf{zero}, \lambda x\ xs\ g.\mathsf{cond}(a\ \mathsf{eq}_A\ x, \mathsf{succ}(g), g)) \\ PERM(k,l) &\equiv \forall a{\in}A.\ noccs\ `\ a\ `\ k = noccs\ `\ a\ `\ l \end{aligned}$$

By defining ord and $noccs$ in terms of primitive recursive functionals, they can be immediately translated into extended first-order logic, allowing properties of ORD and $PERM$ to be more easily derived. If ord and $noccs$ were written using rec or letrec, then their conversion rules could be used as axioms instead, since rec and letrec cannot in general be simply typed.

A sorted list is defined as an ordered permutation of the original.

$$SORT(k,l) \equiv ORD(l) \wedge PERM(k,l)$$

Sorting algorithms are specified by the following type

$$\Pi l{:}\mathsf{List}(A).\{x{:}\mathsf{List}(A), SORT(l,x)\}$$

5.3 Verification

Checking that a program inhabits a simple type raises conditions sufficient for termination; checking that a program inhabits a more complete specification raises conditions for correctness as well as termination.

The tactics for type checking reduce goals of the form $a : A$ to sets of correctness conditions. These conditions can be somewhat simplified before they are translated into extended first-order logic and proved. In particular,

- subtypes in hypotheses can be broken down into simple types and predicates (using $\{\}$-*elim*);
- substitutions that appear as hypotheses in a goal, generated by the strengthened type rules for destructors, can be carried out both in the goal's conclusion and in its other hypotheses; and
- induction hypotheses can be specialised to recursive calls that appear in the conclusion or premises of each goal, unless a specialised hypothesis has already been produced by type checking (see §4.4.1).

All three kinds of simplification are illustrated in the following example. Consider a function that takes an element a and an ordered list l as arguments, and inserts a into l in a way that preserves the ordering. It is proved correct by solving the following goal.

1. $\mathsf{lam}\ a.\mathsf{lam}\ l.\mathsf{letrec}\ \ insert\ a\ l\ \ \mathsf{be}$
   $\qquad\qquad \mathsf{lcase}(l, a \bullet [], \lambda x\ xs.\mathsf{cond}(le\ `\ a\ `\ x, a \bullet x \bullet xs, x \bullet (insert\ `\ a\ `\ xs)))$
   $\mathsf{in}\ \ insert\ `\ a\ `\ l\ \ \mathsf{end}\ \ : \Pi a{:}A.\Pi l{:}\{x{:}\mathsf{List}(A), ORD(x)\}.\{x{:}\mathsf{List}(A), SORT(a \bullet l, x)\}$

Applying `typechk_tac` reduces the goal to the following correctness conditions.

1. $\mathit{SORT}(a \bullet [], a \bullet [])$

2. $[\![\, l : \{x{:}\mathsf{List}(A), \mathit{ORD}(x)\};\;\; l = h \bullet t \,]\!] \;\Longrightarrow\; \mathit{ORD}(t)$

3. $l = h \bullet t \;\Longrightarrow\; \langle a, t\rangle \;\prec_{?R}\; \langle a, l\rangle$

4. $[\![\, a \le h;\;\; l : \{x{:}\mathsf{List}(A), \mathit{ORD}(x)\};\;\; l = h \bullet t \,]\!] \;\Longrightarrow\; \mathit{SORT}(a \bullet h \bullet t, a \bullet h \bullet t)$

5. $[\![\, \forall u{:}A.\forall v{:}\{x{:}\mathsf{List}(A), \mathit{ORD}(x)\}.\, \langle u, v\rangle \;\prec_{?R}\; \langle a, l\rangle \;\supset$
   $\mathit{insert}\,`\,u\,`\,v : \{x{:}\mathsf{List}(A), \mathit{SORT}(u \bullet v, x)\};$
   $\neg a \le h;\;\; l : \{x{:}\mathsf{List}(A), \mathit{ORD}(x)\};\;\; l = h \bullet t \,]\!] \;\Longrightarrow$
   $\mathit{SORT}(a \bullet h \bullet t, h \bullet (\mathit{insert}\,`\,a\,`\,t))$

In goals (2) and (4), the subtype $l : \{x{:}\mathsf{List}(A), \mathit{ORD}(x)\}$ is broken down with $\{\}$-*elim*. In goals (2), (3) and (4), the hypothesis $l = h \bullet t$ is used to substitute $h \bullet t$ for l in other hypotheses and in the conclusion. In (5), the induction hypothesis is instantiated with the call $\mathit{insert}\,`\,a\,`\,t$. This leaves the following set of simplified correctness conditions.

1. $\mathit{SORT}(a \bullet [], a \bullet [])$

2. $\mathit{ORD}(h \bullet t) \;\Longrightarrow\; \mathit{ORD}(t)$

3. $\langle a, t\rangle \;\prec_{?R}\; \langle a, h \bullet t\rangle$

4. $[\![\, a \le h;\;\; \mathit{ORD}(h \bullet t) \,]\!] \;\Longrightarrow\; \mathit{SORT}(a \bullet h \bullet t, a \bullet h \bullet t)$

5. $[\![\, \mathit{SORT}(a \bullet t, \mathit{insert}\,`\,a\,`\,t);\;\; \neg a \le h;\;\; \mathit{ORD}(h \bullet t) \,]\!] \;\Longrightarrow$
   $\mathit{SORT}(a \bullet h \bullet t, h \bullet (\mathit{insert}\,`\,a\,`\,t))$

In principle, these simplifications could be included in a tactic for type checking, so that specifications of correctness are reduced to sets of simplified correctness conditions in a single step. Moreover, if theory interpretation was implemented, these conditions could be automatically translated into first-order logic. Consider such a tactic for automatically generating a set of correctness conditions in first-order logic. The following goal states that an algorithm meets the specification for division given earlier.

1. $\mathsf{lam}\; n.\mathsf{lam}\; d.\; \mathsf{letrec}\; \mathit{div}\; n\; d\; \mathsf{be}\; \mathsf{cond}(\mathit{lt}\,`\,n\,`\,d, \mathsf{zero}, \mathsf{succ}(\mathit{div}\,`\,(n-d)\,`\,d))$
   $\mathsf{in}\;\; \mathit{div}\,`\,n\,`\,d\;\; \mathsf{end}$
   $: \Pi n{:}\mathsf{Nat}.\Pi d{:}\{x{:}\mathsf{Nat}, \neg x = \mathsf{zero}\}.\{q{:}\mathsf{Nat}, \mathit{DIV}(n, d, q)\}$

It would be reduced by such a tactic to the following correctness conditions in first-order logic.

1. $n < d \;\Longrightarrow\; \exists\, r.\; r < d \;\wedge\; n = \mathsf{zero} \times d + r$

2. $[\![\, \neg n < d;\;\; \exists\, r.\; r < d \;\wedge\; n - d = q \times d + r \,]\!] \;\Longrightarrow\; \exists\, r.\; r < d \;\wedge\; n = \mathsf{succ}(q) \times d + r$

3. $\neg d = \mathsf{zero} \;\Longrightarrow\; \langle n - d, d\rangle \;\prec_{?R}\; \langle n, d\rangle$

5.4 Derivation

As type checking continues only as far as a term is instantiated, programs can be instantiated in a stepwise fashion during proof. Again, consider the example of subtractive division. Derivation starts with the following goal.

1. $?a \;:\; \Pi n{:}\mathsf{Nat}.\Pi d{:}\{x{:}\mathsf{Nat}, \neg x = \mathsf{zero}\}.\{q{:}\mathsf{Nat}, \mathit{DIV}(n, d, q)\}$

The tactic `letrec_tac` instantiates the program

$$?a \leftarrow \mathsf{lam}\ n.\mathsf{lam}\ d.\mathsf{letrec}\ \mathit{div}\ \ n\ d\ \mathsf{be}\ ?b(n, d, \mathit{div})\ \mathsf{in}\ \mathit{div}\ \text{‘}\ n\ \text{‘}\ d\ \mathsf{end}$$

and generates an induction hypothesis, leaving the following goal.

1. $\forall u{:}\mathsf{Nat}.\forall v{:}\{x{:}\mathsf{Nat}, \neg x = \mathsf{zero}\}.\ \langle u, v\rangle \ \prec_{?R}\ \langle n, d\rangle \ \supset$
   $\qquad\qquad \mathit{div}\ \text{‘}\ u\ \text{‘}\ v : \{x{:}\mathsf{Nat}, \mathit{DIV}(u, v, x)\} \implies$
   $?b(n, d, \mathit{div}) \;:\; \{x{:}\mathsf{Nat}, \mathit{DIV}(n, d, x)\}$

An obvious approach, according to Manna and Waldinger [34], is to consider solving the first conjunct in the predicate DIV, namely $n < d$. Partially instantiating the program using `itypechk_tac "cond(lt‘n‘d,?t,?u)" [] 1` generates the following subgoals.

1. $n < d \implies ?t(n, d, \mathit{div}) \;:\; \{x{:}\mathsf{Nat}, \mathit{DIV}(n, d, x)\}$

2. $\neg n < d \implies ?u(n, d, \mathit{div}) \;:\; \{x{:}\mathsf{Nat}, \mathit{DIV}(n, d, x)\}$

The first branch is further instantiated by `itypechk_tac "zero" [] 1`, giving

1. $n < d \implies \exists r{:}\mathsf{Nat}.(r < d) \wedge (n = \mathsf{zero} \times d + r)$

2. $\neg n < d \implies ?u(n, d, \mathit{div}) \;:\; \{x{:}\mathsf{Nat}, \mathit{DIV}(n, d, x)\}$

Choosing $r = n$, (1) becomes $n < d \implies n < d$ and $n = \mathsf{zero} \times d + n$, which are clearly solvable by arithmetic reasoning. The constructor `succ` is the only other possible result, and with some insight the appropriate recursion can be seen. Applying `itypechk_tac "succ(div‘(m-n)‘n)" []` gives

1. $[\![\ \neg n \le d;\ \ \mathit{DIV}(n - d, d, \mathit{div}\ \text{‘}\ (n - d)\ \text{‘}\ d)\ ]\!] \implies \mathit{DIV}(n, d, \mathsf{succ}(\mathit{div}\ \text{‘}\ (n - d)\ \text{‘}\ d))$

2. $\neg d = \mathsf{zero} \implies \langle n - d, d\rangle < \langle n, d\rangle$

which may be solved by arithmetic reasoning.

Whether this example convincingly syntheses an algorithm is not relevant—though many papers have been written with such claims. The important point is that the outline of an algorithm may be considered, and the proof obligations inherent in it examined. If these prove satisfactory, then further instantiations can be made, and if necessary retracted, until the program is complete.

5.5 Proving Correctness Conditions

Proving termination, verifying correctness and deriving correct algorithms all raise correctness conditions. The major effort in any proof of correctness is in proving these conditions. In fact, the preliminary steps of correctness proofs are so straightforward that automation is feasible (see §6.2.3).

For convenience, correctness conditions are proved in first-order logic extended with computational types, and then lifted up to CCL using the interpretations described in §4.5. In first-order logic, standard theorem proving techniques are used for reasoning by induction (`ind_tac`), rewriting (`asm_simp_tac` and `asm_simp_case_tac`) and simple logical inference (`fast_tac`).

5.6 An Extended Example: Unification

In 1965, Robinson [58] described an algorithm to unify two expressions. If the expressions have a common instance then the algorithm generates a substitution to yield this instance, otherwise it indicates that the expressions have no such instance. The algorithm was called *unification* and has become central to logic programming and theorem proving. Much work has been done on subsequent improvements [38]; the original algorithm is considered here.

Unification is a good choice for an extended example. Although logically complex, it is a fairly small piece of code that can be succinctly presented and should be familiar to many in this field. It has previously been used as an example by Manna and Waldinger [35], who produced a derivation by hand using their deductive tableau system, and subsequently by Paulson [51], who verified the algorithm on machine using LCF. It has also been synthesised as a logic program by Eriksson [18]. In particular, it is a good choice for the logic CCL, as the proof of termination is rather delicate.

First, a formal specification of the algorithm is given. Then, the first-order theories that arise from this specification are described. Finally, a derivation of the algorithm is considered, and the correctness conditions proved. The majority of lemmas used in establishing correctness are proved by induction followed by rewriting and some propositional reasoning. Most are, therefore, listed without any detail of the proofs. The definitions used to formalise unification are fairly close to Paulson, except for the treatment of termination, which follows Manna and Waldinger. This is not intended to be a full description of the formal derivation of unification (which would needlessly repeat much of Manna and Waldinger's paper), but rather an example that illustrates how proofs of correctness are handled in CCL.

5.6.1 A Formal Specification of Correctness

In the following development, programs are defined by primitive recursion over particular data types. To simplify the presentation, only conversion rules are given rather than complete definitions.

For simplicity, abstract terms are considered as binary trees. This is sufficient to represent Lisp S-expressions and combinator expressions. A term is

- a constant a, b, c, or
- a variable x, y, z, or

- a pair of terms (t, u).

In CCL, the types of constants and variables are taken as parameters of the extended example, and for each an equality function is assumed (eq_{const} and eq_{var} respectively). Terms are formalised as the inductive type Term.

$$\mathsf{Term} \equiv \mu X.\, \mathsf{Const} + \mathsf{Var} + X \times X$$

with constructors const, var and comb respectively and destructors $\mathsf{termcase}$ and $\mathsf{termrec}$. The usual rules for typing, conversion, congruence, the freeness of constructors and induction are derived.

The set of variables in a term, $\mathsf{vars} :: \iota \Rightarrow \iota$, is defined inductively over the structure of terms so that the following conversions hold.

$$\begin{aligned} &\mathsf{vars}(\mathsf{const}(c)) &&= \varnothing \\ &\mathsf{vars}(\mathsf{var}(v)) &&= \{ v \} \\ &\mathsf{vars}(\mathsf{comb}(t, u)) &&= \mathsf{vars}(t) \cup \mathsf{vars}(u) \end{aligned}$$

Note that this uses the finite set theory described in §4.5.3.

The infix, boolean valued functions $\mathsf{occs} :: \iota \Rightarrow \iota \Rightarrow \iota$ and $\mathsf{occseq} :: \iota \Rightarrow \iota \Rightarrow \iota$ are defined so that t occs u evaluates to true iff t strictly occurs in u, and t occseq u evaluates to true iff t occurs in or equals u. The following conversions are immediate consequences of the definitions.

$$\begin{aligned} &t\ \mathsf{occs}\ \mathsf{const}(c) &&= \mathsf{false} \\ &t\ \mathsf{occs}\ \mathsf{var}(v) &&= \mathsf{false} \\ &t\ \mathsf{occs}\ \mathsf{comb}(u, v) &&= (t\ \mathsf{occseq}\ u)\ \mathsf{or}\ (t\ \mathsf{occseq}\ v) \\ &t\ \mathsf{occseq}\ u &&= (t\ \mathsf{eq}_{term}\ u)\ \mathsf{or}\ (t\ \mathsf{occs}\ u) \end{aligned}$$

where or is an infix boolean valued functions corresponding to the predicate $\vee$, namely $\mathsf{lam}\ a.\mathsf{lam}\ b.\mathsf{cond}(a, \mathsf{true}, b)$.

Constants are fixed; *variables* can be *replaced* by terms. Write t/x for the replacement of x by t. A *substitution* θ is a finite set of replacements

$$\{t_1/x_1, \ldots t_n/x_n\}$$

in which the variable names (x_i) are distinct and no replacement is trivial (x_i/x_i). *Applying* a substitution θ to a term t simultaneously replaces every occurrence in t of each variable that is in θ with its corresponding replacement term. The *domain* of a substitution is the set of variables that are so replaced; the *range* of a substitution is the set of variables that appear in the replacement terms. For example, applying the substitution $\{\mathsf{const}(c)/x, \mathsf{var}(x)/y\}$, whose domain is the set $\{ x,\ y \}$ and whose range is $\{ x \}$, to the term

$$\mathsf{comb}(\mathsf{comb}(\mathsf{const}(a), \mathsf{var}(x)), \mathsf{comb}(\mathsf{var}(y), \mathsf{var}(z)))$$

generates the term

$$\mathsf{comb}(\mathsf{comb}(\mathsf{const}(a), \mathsf{const}(c)), \mathsf{comb}(\mathsf{var}(x), \mathsf{var}(z)))$$

Note that variables that do not occur in the domain remain unchanged, and as in this case the domain and range of a substitution need not be distinct.

Following Paulson [51], substitutions are represented by association lists rather than sets, as this is more convenient for the proof of correctness. They are formally represented by the inductive type Alist.

$$\mathsf{Alist} \;\equiv\; \mu\, X .\, \mathsf{Unit} \;+\; \mathsf{Var} \times \mathsf{Term} \times X$$

with constructors anil and acons respectively and destructor alrec. Application is defined in two stages. First, the application of a substitution to a variable is defined by the function $\mathsf{assoc} :: \iota \Rightarrow \iota \Rightarrow \iota$. If variable v appears in substitution θ then $\mathsf{assoc}(v, \theta)$ returns the term corresponding to the first occurrence, otherwise it returns the default term $\mathsf{var}(v)$. The following conversions hold.

$$\begin{aligned} \mathsf{assoc}(v, \mathsf{anil}) &= \mathsf{var}(v) \\ \mathsf{assoc}(v, \mathsf{acons}(w, t, \theta)) &= \mathsf{cond}(v \;\mathsf{eq}_{var}\; w, t, \mathsf{assoc}(v, \theta)) \end{aligned}$$

Secondly, the application of a substitution to a term is inductively defined over terms by the infix function $\lhd :: \iota \Rightarrow \iota \Rightarrow \iota$, so that the following conversions hold.

$$\begin{aligned} \mathsf{const}(c) \lhd \theta &= \mathsf{const}(c) \\ \mathsf{var}(v) \lhd \theta &= \mathsf{assoc}(v, \theta) \\ \mathsf{comb}(t, u) \lhd \theta &= \mathsf{comb}(t \lhd \theta, u \lhd \theta) \end{aligned}$$

This is one possible representation of the informal description above. Multiple occurrences of variables in substitutions are effectively removed, since assoc considers only the first occurrence. The choice of default term for assoc ensures that a trivial substitution behaves on application in the same way as an empty substitution.

Further term-formers over substitutions (for domain, range and equality) are defined in such a way that this representation of substitutions correctly interprets the informal description above. The domain of a substitution, $\mathsf{dom} :: \iota \Rightarrow \iota$, is defined to exclude trivial substitutions, so that the following conversions hold.

$$\begin{aligned} &\mathsf{dom}(\mathsf{anil}) &&= \varnothing \\ &\mathsf{dom}(\mathsf{acons}(v, t, \theta)) &&= \mathsf{cond}(\mathsf{var}(v) \;\mathsf{eq}_{var}\; t, \mathsf{dom}(\theta) - \{\, v \,\}, v \odot \mathsf{dom}(\theta)) \end{aligned}$$

The range of a substitution, $\mathsf{range} :: \iota \Rightarrow \iota$, is defined in terms of its domain using set replacement (see §4.5).

$$\mathsf{range}(\theta) \;\equiv\; \bigcup\{y \mid x \in \mathsf{dom}(\theta) \;,\; y = \mathsf{vars}(x \lhd \theta)\}$$

Equality between substitutions, $=_{subst} :: \iota \Rightarrow \iota \Rightarrow o$, is defined extensionally.

$$\theta =_{subst} \phi \;\;\equiv\;\; \forall\, t{:}\mathsf{Term}.\, t \lhd \theta = t \lhd \phi$$

These definitions permit the following congruences to be derived.

$$\frac{v_1 = v_2 \qquad \theta_1 =_{subst} \theta_2}{\mathsf{assoc}(v_1, \theta_1) =_{subst} \mathsf{assoc}(v_2, \theta_2)}$$

$$\frac{t_1 = t_2 \qquad \theta_1 =_{subst} \theta_2}{t_1 \lhd \theta_1 = t_2 \lhd \theta_2} \qquad\qquad \frac{\theta_1 =_{subst} \theta_2}{\mathsf{dom}(\theta_1) =_{set} \mathsf{dom}(\theta_2)}$$

This ensures that the formal representation captures the intended meaning, and allows the first-order theory `SUBST` in §5.6.2 to be interpreted. Note that $=_{subst}$ is not a congruence relation with respect to the term-former alrec, which is therefore omitted in the theory `SUBST`.

The composition of two substitutions is such that, for all terms, applying the composition is the same as applying the first substitution and then applying the second. Formally, composition is defined as the infix function $\diamond :: \iota \Rightarrow \iota \Rightarrow \iota$, so that the following conversions hold.

$$\mathsf{anil} \diamond \theta = \theta$$
$$\mathsf{acons}(v, t, \phi) \diamond \theta = \mathsf{acons}(v, t \lhd \theta, \phi \diamond \theta)$$

The definitions leading to the notion of a most general, idempotent unifier are now formally introduced. A substitution θ *unifies* two terms t and u iff applying the substitution makes the terms agree.

$$\mathit{Unifies}(\theta, t, u) \;\equiv\; (t \lhd \theta = u \lhd \theta)$$

There may be many unifiers for a pair of terms. For example, the terms $\mathsf{var}(x)$ and $\mathsf{var}(y)$ can be unified by the substitutions $\{\mathsf{var}(x)/y\}$, $\{\mathsf{var}(y)/x\}$, or $\{t/x, t/y\}$ for any term t. A substitution θ is *more general* than a substitution ϕ iff ϕ can be expressed as an instance of θ composed with some substitution ψ.

$$\theta \gg \phi \;\equiv\; \exists\, \psi{:}\mathsf{Alist}. \phi =_{subst} \theta \diamond \psi$$

A unifier θ of terms t and u is *most general* iff it is more general than every unifier of t and u.

$$\mathit{MGU}(\theta, t, u) \;\equiv\; \mathit{Unifies}(\theta, t, u) \;\wedge\; (\forall\, \phi{:}\mathsf{Alist}.\, \mathit{Unifies}(\phi, t, u) \supset \theta \gg \phi)$$

From this definition, every unifier of the terms t and u can be obtained from a most general unifier by composition with a further substitution. A substitution θ is *idempotent* iff it remains unchanged when composed with itself.

$$\mathit{Idem}(\theta) \;\equiv\; \theta \diamond \theta =_{subst} \theta$$

For the algorithm to behave correctly, it must find a most general unifier, if one exists; for the proof of correctness to succeed, the unifier must also be idempotent.

$$\mathit{BestUnifier}(\theta, t, u) \;\equiv\; \mathit{MGU}(\theta, t, u) \wedge \mathit{Idem}(\theta)$$

The absence of a unifier is indicated by using the following data type for optional results.

$$\mathsf{Opt}(A) \;\equiv\; \mathsf{Unit} \;+\; \mathsf{S}(A)$$

with constructors none and some respectively and destructor $\mathsf{optcase}$. The result a of trying to unify the terms t and u is defined as

$$\begin{aligned}\mathit{TryBestUnifier}(a, t, u) \;\equiv\; & (a = \mathsf{none} \;\wedge\; \forall\, \theta{:}\mathsf{Alist}.\neg\, \mathit{Unifies}(\theta, t, u)) \;\vee \\ & (\exists\, \theta{:}\mathsf{Alist}.a = \mathsf{some}(\theta) \;\wedge\; \mathit{BestUnifier}(\theta, t, u))\end{aligned}$$

Finally, a program for unification is specified by the type

$$\Pi t{:}\mathsf{Term}.\Pi u{:}\mathsf{Term}.\{a{:}\mathsf{Opt}(\mathsf{Alist}), \mathit{TryBestUnifier}(a, t, u)\}$$

5.6.2 First-Order Theories

Each data type defined in CCL interprets a meta-type in FOL. The data types introduced for unification (Term, Alist and Opt) lead to extensions of FOL (TERM, SUBST and OPT respectively). The interpretation of TERM and OPT are straightforward. The interpretation of SUBST uses the representation type Alist, and maps $= :: \mathit{subst} \Rightarrow \mathit{subst} \Rightarrow o$ to $=_{subst}$ in a similar way to the interpretation of finite set theory in §4.5.

TERM

TERM introduces the meta-type *term* with constants const, var, comb and termrec, and axioms for the freeness of constructors, induction and conversion for termrec. There is a straightforward interpretation of TERM by the type Term, in which $= :: \mathit{term} \Rightarrow \mathit{term} \Rightarrow o$ maps to $= :: \iota \Rightarrow \iota \Rightarrow o$.

In TERM, the constants

$$\begin{aligned} \mathsf{vars} &:: \mathit{term} \Rightarrow \mathit{var}\ \mathit{set} \\ \mathsf{occs} &:: \mathit{term} \Rightarrow \mathit{term} \Rightarrow \mathit{bool} \\ \mathsf{occseq} &:: \mathit{term} \Rightarrow \mathit{term} \Rightarrow \mathit{bool} \end{aligned}$$

are defined as

$$\begin{aligned} \mathsf{vars}(t) &\equiv \mathsf{termrec}(t, \lambda c.\varnothing, \lambda v.\{\, v \,\}, \lambda x\ y\ g\ h.g \cup h) \\ t\ \mathsf{occs}\ u &\equiv \mathsf{termrec}(u, \lambda c.\mathsf{false}, \lambda v.\mathsf{false}, \lambda x\ y\ g\ h.((t\ \mathsf{eq}\ x)\ \mathsf{or}\ g)\ \mathsf{or}\ ((u\ \mathsf{eq}\ x)\ \mathsf{or}\ h)) \\ t\ \mathsf{occseq}\ u &\equiv (t\ \mathsf{eq}\ u)\ \mathsf{or}\ (t\ \mathsf{occs}\ u) \end{aligned}$$

The conversion rules of §5.6.1 are immediately derived from these definitions.

Using induction and the freeness of constructors, theorems are derived for the well-foundedness of terms

$$\neg\mathsf{comb}(t,u) = t \qquad\qquad \neg\mathsf{comb}(t,u) = u$$

the transitivity of occs

$$t\ \mathsf{occs}\ u = \mathsf{true} \supset u\ \mathsf{occs}\ v = \mathsf{true} \supset t\ \mathsf{occs}\ v = \mathsf{true}$$

and, using transitivity, the irreflexivity of occs

$$t\ \mathsf{occs}\ t = \mathsf{false}$$

By rewriting, the following equivalences for vars are derived.

$$\begin{aligned} v \in \mathsf{vars}(\mathsf{var}(w)) &\leftrightarrow w = v \\ v \in \mathsf{vars}(t) &\leftrightarrow \mathsf{var}(v)\ \mathsf{occseq}\ t = \mathsf{true} \end{aligned}$$

OPT

OPT introduces the meta-type *opt* with constants none, some and optcase, and axioms for the freeness of constructors, induction and conversion for optcase. There is a straightforward interpretation of OPT by the type Opt, in which $= :: \alpha\ \mathit{opt} \Rightarrow \alpha\ \mathit{opt} \Rightarrow o$ maps to $= :: \iota \Rightarrow \iota \Rightarrow o$.

$$\neg\mathsf{anil} \;=\; \mathsf{acons}(v,t,\theta) \qquad \frac{P(\mathsf{anil}) \qquad \begin{matrix}\left[\; P(X) \;\right]_X \\ P(\mathsf{acons}(v,t,X))\end{matrix}}{P(\theta)}$$

$$\theta = \phi \;\leftrightarrow\; (\forall t.\; t \lhd \theta \;=\; t \lhd \phi)$$

$$\begin{aligned}\mathsf{assoc}(v,\mathsf{anil}) &= \mathsf{var}(v) & \mathsf{assoc}(v,\mathsf{acons}(w,t,\theta)) &= \mathsf{cond}(v \;\mathsf{eq}\; w, t, \mathsf{assoc}(v,\theta)) \\ \mathsf{anil} \diamond \theta &= \theta & \mathsf{acons}(v,t,\phi) \diamond \theta &= \mathsf{acons}(v, t \lhd \theta, \phi \diamond \theta)\end{aligned}$$

$$\begin{aligned}\mathsf{dom}(\mathsf{anil}) &= \varnothing \\ \mathsf{dom}(\mathsf{acons}(v,t,\theta)) &= \mathsf{cond}(\mathsf{var}(v) \;\mathsf{eq}\; t, \mathsf{dom}(\theta) - \{\, v \,\}, v \odot \mathsf{dom}(\theta))\end{aligned}$$

Axioms

$$\begin{aligned}t \lhd \theta &\;\equiv\; \mathsf{termrec}(t, \lambda c.\mathsf{const}(c), \lambda v.\mathsf{assoc}(v,\theta), \lambda x\; y\; g\; h.\mathsf{comb}(g,h)) \\ \mathsf{range}(\theta) &\;\equiv\; \bigcup(\{y \mid x \in \mathsf{dom}(\theta)\;,\; y = \mathsf{vars}(\mathsf{var}(x) \lhd \theta)\})\end{aligned}$$

Definitions

Figure 5.1: *A Theory of Substitutions*

SUBST

`SUBST` introduces the meta-type *subst* with constants

$$\begin{aligned}\mathsf{anil} &:: \mathit{subst} \\ \mathsf{acons} &:: \mathit{var} \Rightarrow \mathit{term} \Rightarrow \mathit{subst} \Rightarrow \mathit{subst} \\ \mathsf{assoc} &:: \mathit{var} \Rightarrow \mathit{subst} \Rightarrow \mathit{term} \\ \diamond &:: \mathit{subst} \Rightarrow \mathit{subst} \Rightarrow \mathit{subst} \\ \mathsf{dom} &:: \mathit{subst} \Rightarrow \mathit{var\ set}\end{aligned}$$

and the axioms shown in Figure 5.1 for induction, extensional equality and conversion, and definitions for substitution and range.

$$\begin{aligned}\lhd &:: \mathit{term} \Rightarrow \mathit{subst} \Rightarrow \mathit{subst} \\ \mathsf{range} &:: \mathit{subst} \Rightarrow \mathit{var\ set}\end{aligned}$$

There is an interpretation of `SUBST` in which the meta-type *subst* is mapped on to the representation type Alist and $= :: \mathit{subst} \Rightarrow \mathit{subst} \Rightarrow o$ is mapped on to $=_{subst}$, and the necessary congruences for $=_{subst}$ all hold.

In `SUBST`, conversion rules for $\lhd$ are immediately derived. Using these and the other rewrite rules of `SUBST`, the facts shown in Figure 5.2 are derived. In this case, the Isabelle proof scripts are shown in Figure 5.4; hereafter, they are omitted. For the domain and range of substitutions, the facts in Figure 5.3 are derived.

$$\begin{array}{rcl}
t \lhd \phi \diamond \theta & = & t \lhd \phi \lhd \theta \\
q \diamond \phi \diamond \theta & = & q \diamond (\phi \diamond \theta) \\
\mathsf{acons}(v, \mathsf{var}(v) \lhd \theta, \theta) & = & \theta \\
t \ \mathsf{occs}\ u = \mathsf{true} & \supset & t \lhd \theta \ \mathsf{occs}\ u \lhd \theta = \mathsf{true} \\
\mathsf{var}(v) \ \mathsf{occs}\ t = \mathsf{false} & \supset & t \lhd \mathsf{acons}(v, t \lhd \theta, \theta) \ = \ t \lhd \theta \\
t \lhd \phi \ = \ t \lhd \theta & \leftrightarrow & (\forall\, v.\ v \in \mathsf{vars}(t) \ \supset \ \mathsf{var}(v) \lhd \phi \ = \ \mathsf{var}(v) \lhd \theta) \\
\neg v \in \mathsf{vars}(t) & \supset & t \lhd \mathsf{acons}(v, u, \theta) \ = \ t \lhd \theta \\
v \in \mathsf{vars}(t) & \supset & w \in \mathsf{vars}(t \lhd \mathsf{acons}(v, \mathsf{var}(w), \theta))
\end{array}$$

Figure 5.2: *Facts about Substitutions*

$$\begin{array}{rcl}
v \in \mathsf{dom}(s) & \leftrightarrow & \neg \mathsf{var}(v) \lhd s \ = \ \mathsf{var}(v) \\
v \in \mathsf{range}(s) & \supset & (\exists\, w.\ w \in \mathsf{dom}(s) \ \wedge \ v \in \mathsf{vars}(\mathsf{var}(w) \lhd s)) \\
t \lhd s \ = \ t & \leftrightarrow & \mathsf{dom}(s) \cap \mathsf{vars}(t) = \varnothing \\
\multicolumn{3}{c}{v \in \mathsf{dom}(s) \ \supset \ \neg v \in \mathsf{range}(s) \ \supset \ \neg v \in \mathsf{vars}(t \lhd s)} \\
\multicolumn{3}{c}{v \in \mathsf{dom}(s) \ \supset \ v \in \mathsf{vars}(t \lhd s) \ \supset \ v \in \mathsf{range}(s)} \\
v \in \mathsf{vars}(t \lhd s) & \supset & v \in \mathsf{range}(s) \ \vee \ v \in \mathsf{vars}(t) \\
\mathsf{dom}(s) \cap \mathsf{range}(s) \ = \ \varnothing & \leftrightarrow & (\forall\, t.\ \mathsf{dom}(s) \cap \mathsf{vars}(t \lhd s) \ = \ \varnothing)
\end{array}$$

Figure 5.3: *Facts about Domains and Ranges*

```
goal subst_thy  "t <| r <> s = t <| r <| s";
by (ind_tac "t" 1);
by (ind_tac "r" 2);
by (ALLGOALS (asm_simp_case_tac [substC2] []));
val subst_comp = result();

goal subst_thy  "q <> r <> s = q <> (r <> s)";
by (ind_tac "q" 1);
by (ALLGOALS (asm_simp_tac [subst_comp] []));
val comp_assoc = result();

goal subst_thy  "acons(w,var(w) <| s,s) = s";
br substeqI 1;
by (ind_tac "t" 1);
by (ALLGOALS (asm_simp_case_tac [substC2,eq_iff1] []));
val acons_trivial = result();

goal subst_thy  "t occs u = true --> t <| s occs u <| s = true";
by (ind_tac "u" 1);
by (ALLGOALS (asm_simp_tac [or_true,eq_iff1] []));
by (fast_tac FOL_cs 1);
val subst_mono = result();

goal subst_thy  "var(v) occs t = false --> t <| acons(v,t <| s,s) = t <| s";
by (cla_case_tac "t = var(v)" 1);
by (res_inst_tac [("P","\%x.~x=var(v) --> var(v) occs x=false --> x<|?t=x<|s")] term_ind 2);
by (ALLGOALS (simp_case_tac [substC2,or_false,eq_iff1,eq_iff2] []));
by (fast_tac FOL_cs 1);
val var_not_occs = result();

goal subst_thy  "t <| r = t <| s <-> (ALL v.v <: vars(t) --> var(v) <| r = var(v) <| s)";
by (ind_tac "t" 1);
by (ALLGOALS (asm_simp_tac [substC2,un_iff] []));
by (ALLGOALS (fast_tac FOL_cs));
val agreement = result();

goal subst_thy  "~ v<: vars(t) --> t <| acons(v,u,s) = t <| s";
by (asm_simp_case_tac [substC2,agreement,eq_iff1] [] 1);
val repl_invariance = result();

goal subst_thy  "v <: vars(t) --> w <: vars(t <| acons(v,var(w),s))";
by (ind_tac "t" 1);
by (ALLGOALS (asm_simp_tac [substC2,eq_refl,un_iff] []));
by (fast_tac FOL_cs 1);
val var_in_subst = result();
```

Figure 5.4: *Isabelle Proofs for Substitutions*

5.6.3 Derivation of Unification

The derivation follows what by now is a familiar pattern. The algorithm is progressively instantiated using `itypechk_tac`, generating a set of correctness conditions, which are then proved in first-order logic. A unification algorithm is derived by solving the goal

1. $?\mathit{unify} \;:\; \Pi t_1{:}\mathsf{Term}.\Pi t_2{:}\mathsf{Term}.\{a{:}\mathsf{Opt}(\mathsf{Alist}),\, \mathit{TryBestUnifier}(a, t_1, t_2)\}$

The most obvious approach is to use `letrec_tac`, which instantiates the general form of a recursive program

$$?\mathit{unify} \;\leftarrow\; \mathsf{lam}\ t_1.\mathsf{lam}\ t_2.\mathsf{letrec}\ \ \mathit{unify}\, t_1\ t_2\ \mathsf{be}\ \ ?b(t_1, t_2, \mathit{unify})\ \mathsf{in}\ \ \mathit{unify}\,`\,t_1\,`\,t_2\ \mathsf{end}$$

and leaves the goal

1. $[\![\, t_1 : \mathsf{Term};\;\; t_2 : \mathsf{Term};\;\; \mathit{IH}\ ]\!] \implies$
   $?b(t_1, t_2, \mathit{unify}) \;:\; \{a{:}\mathsf{Opt}(\mathsf{Alist}),\, \mathit{TryBestUnifier}(a, t_1, t_2)\}$

where *IH* is the induction hypothesis

$$\begin{array}{l} \forall\, u{:}\mathsf{Term}.\forall\, v{:}\mathsf{Term}.\ \ \langle u, v\rangle \prec_{?R} \langle t_1, t_2\rangle \supset \\ \qquad\qquad\qquad \mathit{unify}\,`\,u\,`\,v \;:\; \{a{:}\mathsf{Opt}(\mathsf{Alist}),\, \mathit{TryBestUnifier}(a, u, v)\} \end{array}$$

Now, a little insight is needed. One obvious possibility is case analysis on the first argument t_1. Using the tactic

```
itypechk_tac "termcase(t1,?b,?c,?d)" 1
```

further instantiates the program with

$$?b(t_1, t_2, \mathit{unify}) \;\leftarrow\; \mathsf{termcase}(t_1, ?d(t_1, t_2, \mathit{unify}), ?e(t_1, t_2, \mathit{unify}), ?f(t_1, t_2, \mathit{unify}))$$

and leaves the following goals.

1. $[\![\, t_1 : \mathsf{Term};\;\; t_2 : \mathsf{Term};\;\; \mathit{IH};\;\; c : \mathsf{Const};\;\; t_1 = \mathsf{const}(c)\ ]\!] \implies$
   $?d(t_1, t_2, \mathit{unify}, c) \;:\; \{a{:}\mathsf{Opt}(\mathsf{Alist}),\, \mathit{TryBestUnifier}(a, \mathsf{const}(c), t_2)\}$

2. $[\![\, t_1 : \mathsf{Term};\;\; t_2 : \mathsf{Term};\;\; \mathit{IH};\;\; v : \mathsf{Var};\;\; t_1 = \mathsf{var}(v)\ ]\!] \implies$
   $?e(t_1, t_2, \mathit{unify}, v) \;:\; \{a{:}\mathsf{Opt}(\mathsf{Alist}),\, \mathit{TryBestUnifier}(a, \mathsf{var}(v), t_2)\}$

3. $[\![\, t_1 : \mathsf{Term};\;\; t_2 : \mathsf{Term};\;\; \mathit{IH};\;\; x : \mathsf{Term};\;\; y : \mathsf{Term};\;\; t_1 = \mathsf{comb}(x, y)\ ]\!] \implies$
   $?f(t_1, t_2, \mathit{unify}, x, y) \;:\; \{a{:}\mathsf{Opt}(\mathsf{Alist}),\, \mathit{TryBestUnifier}(a, \mathsf{comb}(x, y), t_2)\}$

The first goal can then be reduced by case analysis on the second argument t_2. Continuing in this fashion, an algorithm for unification can be derived. The interesting part of this derivation is not the instantiation of the algorithm, which is fairly straightforward, but the lemmas that this generates for correctness. The complete program is

```
lam $t_1$.lam $t_2$.letrec  $unify\ t_1\ t_2$ be
   termcase($t_1$,$\lambda c_1$.termcase($t_2$,$\lambda c_2$.cond($c_1$ eq$_{const}$ $c_2$, some(anil), none),
                                   $\lambda v_2$.assign($v_2$, const($c_1$)),
                                   $\lambda x_2\ y_2$.none),
             $\lambda v_1$.assign($v_1$, $t_2$),
             $\lambda x_1\ y_1$.termcase($t_2$,$\lambda c_2$.none,
```

$$\begin{array}{l} \qquad\qquad\qquad\qquad\qquad\qquad \lambda v_2.\mathsf{assign}(v_2, \mathsf{comb}(x_1, y_1)), \\ \qquad\qquad\qquad\qquad\qquad\qquad \lambda x_2\ y_2.\mathsf{optcase}(\mathit{unify}\ `\ x_1\ `\ x_2, \mathsf{none}, \\ \qquad\qquad\qquad\qquad\qquad\qquad\ \ \lambda\theta.\mathsf{optcase}(\mathit{unify}\ `\ (y_1 \lhd \theta)\ `\ (y_2 \lhd \theta), \mathsf{none}, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \lambda\phi.\mathsf{some}(\theta \diamond \phi))))) \\ \mathsf{in}\ \ \mathit{unify}\ `\ t_1\ `\ t_2\ \ \mathsf{end} \end{array}$$

where $\mathsf{assign}(v, t) \equiv \mathsf{cond}(\mathsf{var}(v)\ \mathsf{occs}\ t, \mathsf{none}, \mathsf{some}(\mathsf{acons}(v, t, \mathsf{anil})))$. Using `itypechk_tac` to instantiate $?\mathit{unify}$ with this algorithm, and then simplifying the correctness conditions raised using the tactics of §4.4 generates the following conditions for partial correctness:

C1. $c_1 = c_2 \implies \mathit{TryBestUnifier}(\mathsf{some}(\mathsf{anil}), \mathsf{const}(c_1), \mathsf{const}(c_2))$

C2. $\neg c_1 = c_2 \implies \mathit{TryBestUnifier}(\mathsf{none}, \mathsf{const}(c_1), \mathsf{const}(c_2))$

C3. $\mathit{TryBestUnifier}(\mathsf{assign}(v_2, \mathsf{const}(c_1)), \mathsf{const}(c_1), \mathsf{var}(v_2))$

C4. $\mathit{TryBestUnifier}(\mathsf{none}, \mathsf{const}(c_1), \mathsf{comb}(x_2, y_2))$

C5. $\mathit{TryBestUnifier}(\mathsf{assign}(v_1, t_2), \mathsf{var}(v_1), t_2)$

C6. $\mathit{TryBestUnifier}(\mathsf{none}, \mathsf{comb}(x_1, y_1), \mathsf{const}(c_2))$

C7. $\mathit{TryBestUnifier}(\mathsf{assign}(v_2, \mathsf{comb}(x_1, y_1)), \mathsf{comb}(x_1, y_1), \mathsf{var}(v_2))$

C8. $\mathit{TryBestUnifier}(\mathsf{none}, x_1, x_2) \implies$
$\mathit{TryBestUnifier}(\mathsf{none}, \mathsf{comb}(x_1, y_1), \mathsf{comb}(x_2, y_2))$

C9. $[\![\ \mathit{TryBestUnifier}(\mathsf{some}(\theta), x_1, x_2);\quad \mathit{TryBestUnifier}(\mathsf{none}, y_1 \lhd \theta, y_2 \lhd \theta)\ ]\!] \implies$
$\mathit{TryBestUnifier}(\mathsf{none}, \mathsf{comb}(x_1, y_1), \mathsf{comb}(x_2, y_2))$

C10. $[\![\ \mathit{TryBestUnifier}(\mathsf{some}(\theta), x_1, x_2);\quad \mathit{TryBestUnifier}(\mathsf{some}(\phi), y_1 \lhd \theta, y_2 \lhd \theta)\ ]\!] \implies$
$\mathit{TryBestUnifier}(\mathsf{some}(\theta \diamond \phi), \mathsf{comb}(x_1, y_1), \mathsf{comb}(x_2, y_2))$

and the following conditions for termination:

T1. $\langle x_1, x_2\rangle \prec_{?R} \langle \mathsf{comb}(x_1, y_1), \mathsf{comb}(x_2, y_2)\rangle$

T2. $\mathit{TryBestUnifier}(\mathsf{some}(\theta), x_1, x_2) \implies$
$\langle y_1 \lhd \theta, y_2 \lhd \theta\rangle \prec_{?R} \langle \mathsf{comb}(x_1, y_1), \mathsf{comb}(x_2, y_2)\rangle$

Note that type checking does not instantiate the well-founded ordering $?R$. The form of the goals (T1) and (T2) can be used to suggest an instantiation for $?R$. Manna and Waldinger use the following lexicographic ordering in their proof.

$$\begin{array}{rcl} \langle a_1, a_2\rangle \prec_{\mathrm{M+W}} \langle b_1, b_2\rangle & \equiv & \mathsf{vars}(a_1) \cup \mathsf{vars}(a_2) \subset \mathsf{vars}(b_1) \cup \mathsf{vars}(b_2) \quad \vee \\ & & \quad (\mathsf{vars}(a_1) \cup \mathsf{vars}(a_2) = \mathsf{vars}(b_1) \cup \mathsf{vars}(b_2) \ \wedge \\ & & \quad\ a_1\ \mathsf{occs}\ b_1 = \mathsf{true}) \end{array}$$

Informally, $\langle a_1, a_2\rangle \prec_{\mathrm{M+W}} \langle b_1, b_2\rangle$ holds iff the variables in a_1 and a_2 are a strict subset of those in b_1 and b_2, or the variables are identical and the term a_1 occurs in b_1. In CCL, it is simpler to formalise a slightly weaker relation that holds iff the cardinality of one set of variables is less-than the cardinality of the other, rather than one set actually being contained in the other. $?R$ is instantiated as follows:

$$?R \leftarrow \mathsf{map}(f, \mathsf{lex}(\mathsf{pR}, \mathsf{pR}))$$

where f is a function that maps pairs $\langle x, y\rangle$ to pairs $\langle n, x\rangle$, in which n is the cardinality of the set of variables in x or in y:

$$f \equiv \mathsf{lam}\ p.\mathsf{split}(p, \lambda x\ y.\langle \mathsf{card}(\mathsf{vars}(x) \cup \mathsf{vars}(y)), x\rangle)$$

With this instantiation for $?R$, the termination conditions are simplified to the following goals.

T1. $\mathit{WFD}(x_1, x_2, \mathsf{comb}(x_1, y_1), \mathsf{comb}(x_2, y_2))$

T2. $\mathit{TryBestUnifier}(\mathsf{some}(\theta), x_1, x_2) \implies$
$\mathit{WFD}(y_1 \lhd \theta, y_2 \lhd \theta, \mathsf{comb}(x_1, y_1), \mathsf{comb}(x_2, y_2))$

where

$$\begin{aligned}\mathit{WFD}(a_1, a_2, b_1, b_2) \equiv\ & \mathsf{card}(\mathsf{vars}(a_1) \cup \mathsf{vars}(a_2)) \prec_{\mathsf{pR}} \mathsf{card}(\mathsf{vars}(b_1) \cup \mathsf{vars}(b_2)) \quad \vee \\ & (\mathsf{card}(\mathsf{vars}(a_1) \cup \mathsf{vars}(a_2)) = \mathsf{card}(\mathsf{vars}(b_1) \cup \mathsf{vars}(b_2)) \wedge \\ & a_1 \prec_{\mathsf{pR}} b_1)\end{aligned}$$

The conditions for partial correctness and termination are now proved in first-order logic, using the facts already derived for substitutions and their domains and ranges.

For partial correctness, the conditions all follow from the lemmas of Figure 5.5. In particular, (C3), (C5) and (C7) are all instances of the following general result for unifying variables.

$$\mathit{BestUnifier}(\mathsf{assign}(v, t), \mathsf{var}(v), t)$$

Only (C10) is awkward, requiring careful direction of the rewriting.

Similarly, the conditions for termination are proved from the following lemmas.

$$\begin{aligned} x \subset y &\supset \mathsf{card}(x) < \mathsf{card}(y) \\ \mathit{BestUnifier}(\theta, t, u) &\supset \mathsf{dom}(\theta) \subseteq \mathsf{vars}(t) \cup \mathsf{vars}(u) \\ \mathit{BestUnifier}(\theta, t, u) &\supset \mathsf{range}(\theta) \subseteq \mathsf{vars}(t) \cup \mathsf{vars}(u)\end{aligned}$$

though the last two require some effort to prove.

The complete derivation uses forty two lemmas. As an example, unification is a little misleading; proving the correctness of this algorithm is far more complex than it is for many other programs. But what it does clearly show, is how theory can be developed in a natural way and an algorithm specified, how type checking uses an algorithm to generate correctness conditions, and how these can be solved in typed first-order logic using induction and rewriting.

$$
\begin{array}{rcl}
\mathsf{anil} \gg \theta & & \\
 & & \mathit{Idem}(\mathsf{anil}) \\
\mathit{MGU}(\theta, t, u) & \leftrightarrow & (\forall\, \phi.\ \mathit{Unifies}(\phi, t, u) \leftrightarrow \theta \gg \phi) \\
\mathit{BestUnifier}(\theta, t, u) & \leftrightarrow & \mathit{Idem}(\theta)\ \wedge\ \mathit{Unifies}(\theta, t, u)\ \wedge \\
 & & \qquad\qquad (\forall\, \phi.\ \mathit{Unifies}(\phi, t, u) \supset \theta \gg \phi) \\
\mathit{Idem}(\theta) & \leftrightarrow & \mathsf{dom}(\theta) \cap \mathsf{range}(\theta)\ =\ \varnothing \\
\mathsf{vars}(v)\ \mathsf{occs}\ t\ =\ \mathsf{false} & \supset & \mathit{Idem}(\mathsf{acons}(v, t, \mathsf{anil})) \\
\mathit{BestUnifier}(\theta, t, u) & \leftrightarrow & \mathit{BestUnifier}(\theta, u, t)
\end{array}
$$

Figure 5.5: *Lemmas for Partial Correctness*

Chapter 6

Conclusion

This chapter draws conclusions from the work presented in the dissertation and suggests areas for future research.

6.1 Results

The main contribution of this work is that the theory was actually implemented and formal correctness proofs carried out on machine. Although many features of programming languages remain to be addressed (e.g. modules, polymorphism and concurrency), proving the correctness of programs in CCL has given some indication of what is demanded of a system for formal correctness. It is clear from this work that

- although appealing from some philosophical standpoints, intuitionism is not necessary for a computational logic;
- dependent types are advantageous for expressing termination and for generating correctness conditions by type checking;
- general recursion can be reasoned about in a natural way using well-founded induction, and termination can be separated from partial correctness;
- interleaving programming and verification is an effective approach to derivation; it is far easier to write programs that should implement specifications than to prove that specifications can be satisfied; and
- an evaluation semantics is a suitable basis for a computational logic.

These last two points are considered further.

6.1.1 Interleaving Programming with Verification

In CCL, programs can serve as templates to direct proofs of correctness. As there is a single type rule for each term-former in the language, a goal of the form $a : A$ is broken down uniquely by type checking. In program verification, the term a is fully instantiated, and type checking immediately reduces the goal $a : A$ to a set of correctness conditions. In program derivation, the term a is progressively instantiated during the proof. At each stage, a program fragment is suggested for a, which by type checking generates a further set of correctness conditions and unfulfilled subspecifications. Derivation proceeds by interleaving programming and verification. The form of correctness conditions generated by possible algorithms can be easily examined; when these conditions appear unsolvable, backtracking allows the instantiation to be retracted and an alternative considered.

Experience with the interactive style of development encouraged by ML suggests an approach that lies between verification and derivation. In ML, individual functions (or parts thereof) are written and then passed to the type checker. Errors picked up by type checking can be corrected and the process repeated until the function is well-typed. Similarly, to develop correct code, individual functions are written and then verified. If the correctness conditions indicate errors, these can be corrected and the process repeated. Finally, the function is shown to be correct by proving the correctness conditions.

6.1.2 Denotational and Operational Approaches

In §1.3, a distinction was made between denotational and operational computational logics. Clearly, CCL falls into the latter category. Programs are represented by abstract syntax trees, and reasoning is based on an evaluation relation.

This work began, however, by taking a denotational approach. Terminating programs were represented by functions in ZF set theory, which required annotations for the types of λ-abstractions and for recursion (represented by well-founded recursion). The extra information present in program denotations made them awkward. Moreover, facts were proved about denotations and not the programs themselves; extra work was required to relate these facts to computational behaviour. This required significantly more effort than an operational approach—without any clear benefits.

Although CCL is an operational logic, it does not suffer the limitations of transformation systems since it permits specification. Reasoning about the correctness of programs in any case has an operational character; new theory is often developed hand-in-hand with an implementation, although occasionally theory is sometimes developed for its own sake (e.g. finite set theory is used to reason about the correctness of a unification algorithm in §5.6). But the development of set theory is not unduly awkward, and the implementation it entails would prove useful in other programs. If this style of development did prove to be inconvenient, CCL could be based on higher-order logic instead of first-order logic. So far, however, this has not proved necessary.

6.2 Future Work

The approach taken in deriving CCL from $\mathcal{L}$ can be adopted for other target languages with different evaluation semantics. Experience with the implementation of CCL suggests improvements to the tactics for type checking. In addition, it should be possible to extend the simple tactics described here to provide a more complete environment for the derivation of correct programs.

6.2.1 Other Target Languages

CCL arose directly from the evaluation semantics of its target language $\mathcal{L}$. The same development could be carried out for other functional languages with eager as well as lazy evaluation. For CCL, development was carried out by hand and the resulting calculus implemented as a set of axioms in Isabelle. The effort required to derive the single theory CCL within Isabelle's set theory (or higher-order logic) would outweigh any advantage that this mechanisation might give. But there would be advantages in establishing a general suite of theorems and tactics for developing computational logics from target programming languages. In particular, the consequences of small changes to a target language could be studied more easily.

One possible variation is from a lazy to an eager evaluation strategy. Following the development in Chapter 3, the computational logic CCL$_e$ can be derived from the target language $\mathcal{L}_e$ (identical to $\mathcal{L}$ but with eager evaluation). The eager evaluation relation $\rhd_e$ differs from its lazy counterpart $\rhd$ only in the rules for the constructors s and $\langle\rangle$ (as in languages such as ML, the constructor lam remains lazy).

$$\frac{u \ \rhd_e \ a}{\mathsf{s}(u) \ \rhd_e \ \mathsf{s}(a)} \qquad\qquad \frac{u \ \rhd_e \ a \qquad v \ \rhd_e \ b}{\langle u, v\rangle \ \rhd_e \ \langle a, b\rangle}$$

CCL$_e$ differs from CCL only in the conversion rules involving these eager constructors.

$$\begin{array}{rcl} a\!\downarrow & \Longrightarrow & \mathsf{scase}(\mathsf{s}(a), b, c) = c(a) \\ [\![\, a\!\downarrow;\ b\!\downarrow \,]\!] & \Longrightarrow & \mathsf{split}(\langle a, b\rangle, c) = c(a, b) \end{array}$$

The premises ensure the termination of terms, just as the conversion rule for let is conditional (Theorem 3.8). Using types to ensure termination, as in the conversion rule for let, leads to extra premises in many of the derived rules. For example, the injection rule for pairing in CCL$_e$ is

$$\frac{\langle a, b\rangle = \langle a', b'\rangle \qquad \begin{matrix}\left[\ a = a'\ ;\ \ b = b'\ \right] \\ P\end{matrix} \qquad a : A \qquad a' : A' \qquad b : B \qquad b' : B'}{P}$$

Although this puts a significant overhead on reasoning within CCL$_e$ compared with CCL, reasoning in the corresponding interpreted first-order logic is no more onerous. The examples in this dissertation could be carried out in an eager regime with little extra effort.

6.2.2 Improving Type Checking

In CCL, goals of the form $a : \{x{:}A, P(x)\}$ focus attention on program a in the proposition $P(a)$. Type checking directs the proof of these goals using the program a, generating correctness conditions from specifications. But the type checking strategy described in this dissertation suffers two weaknesses, due to the type rules for local declarations and for constructors. These weaknesses and possible solutions are described; neither solution was implemented.

Firstly, as the rules for local declarations (i.e. let, letfun and letrec) introduce uninstantiated type variables, they are incompatible with a general tactic for type checking (see §4.4.4). The simplest solution to the problematic use of let-*type* is to apply the conversion rule for let before type checking, which has the effect of the following rule.

$$\frac{a : A \qquad b(a) : B}{\mathsf{let}\ x\ \mathsf{be}\ a\ \mathsf{in}\ b(x)\ \mathsf{end} : B}$$

The premises can now be independently proved, but at the cost of type checking each occurrence of a in $b(a)$ separately. If local declarations are used with relatively few substitution instances, then this is a practical solution. This is generally the case, since more frequently used identifiers tend to be declared at top level. In fact, it is worthwhile performing all outstanding reductions in a term before type checking (except those for rec).

Secondly, as the type rules for constructors use only simple types, they cannot be applied to subtypes. For goals of the form $a : \{x{:}A, P(x)\}$, in which the outermost term-former of a is a constructor, $\{\}$-*intr* must be applied before the appropriate type rule. This removes the focussed style of goal, preventing further direction of the proof of $P(a)$ by the program a. For example, type checking the goal

1. $\mathsf{succ}(n) \ : \ \{x{:}\mathsf{Nat}, P(x)\}$

uses $\{\}$-*intr* followed by succ-*type*, leaving the subgoals

1. $n : \mathsf{Nat}$

2. $P(\mathsf{succ}(n))$

Further type checking cannot direct the proof of $P(n)$. Instead, if the derived rule

$$\frac{n \ : \ \{x{:}\mathsf{Nat}, P(\mathsf{succ}(x))\}}{\mathsf{succ}(n) \ : \ \{x{:}\mathsf{Nat}, P(x)\}}$$

is used, then $\{\}$-*intr* is no longer necessary and the resulting goal is

1. $n \ : \ \{x{:}\mathsf{Nat}, P(\mathsf{succ}(x))\}$

Further type checking will direct the proof of $P(\mathsf{succ}(n))$.

For constructors with more than one argument, type judgements are nested; this is possible since $a{:}A$ is just a formula in CCL. Type checking is first directed by one argument and then by the other. For example, the rule $\langle\rangle$-*type*

$$\frac{a : A \qquad b : B(a)}{\langle a, b\rangle \ : \ \Sigma x{:}A.B(x)}$$

is replaced by the following derived rule.

$$\frac{a \ : \ \{x{:}A, \ \ b \ : \ \{y{:}B(x), P(\langle x, y\rangle)\} \ \}}{\langle a, b\rangle \ \ : \ \ \{z{:}\Sigma x{:}A.B(x), P(z)\}}$$

Subtypes are finally removed either by type rules for constructors with no arguments, such as

$$\frac{P(\mathsf{zero})}{\mathsf{zero} : \{x{:}\mathsf{Nat}, P(x)\}}$$

or by the type rules for recursive calls (see §4.4.1).

These rules have been derived within CCL, but a tactic for type checking using them has not been written. If these rules are implemented and used in type checking, it may be more convenient to make all types in CCL subtypes (Nat would then become an abbreviation for $\{x{:}\mathsf{Nat}, \top\}$). This would obviate the need for both simple and subtyped rules for each type-former, and simplify the handling of subtypes in hypotheses.

6.2.3 An Environment for Program Correctness

For the formal synthesis of programs to be practical, theory and code should be developed within the same framework of modules. Extended ML takes this approach [60]. The work done in this dissertation should naturally extend to provide an environment for program synthesis.

Programs in CCL are logical theories. New functions and types are declared by extending a theory with definitions of new constants. Program modules structure these theories in the usual mathematical way; namely by restricting the scope of constants, introducing axioms that are justified by theorems in underlying theories, and declaring abstract data types that are implemented by underlying representation types. Program execution is by inference, using a simple set of rules (an evaluation semantics).

Viewed in this way, programs developed in conventional programming languages are theories in which the only possible inference is evaluation. Transformation systems extend program theories with a predicate for equality, which allows partial evaluation and substitution. The computational logic CCL extends transformation systems with a predicate for type membership, where types are sets of terminating programs, and the connectives of first-order logic, where quantification is over programs and types. This is useful because

- if two programs are equal then they evaluate to results that are equal, and
- if a program is typeable then its evaluation terminates.

Programs are evaluated in the context of a module; similarly, program correctness is asserted and proved in the context of a theory. While it is straightforward to determine the evaluation rules that are available in a particular context, determining the other inference rules that are available is more difficult. CCL can be viewed as a meta-theory in which to interpret extensions to first-order logic with computational types. These extensions can represent the sets of inference rules available in theories. Extensions can also be used to axiomatise behaviour that will not be proved correct.

Imagine an environment for deriving correct programs in which a single module system structures program declarations, extensions to first-order logic, and axiomatisations of program behaviour. The correctness of a program in some module is asserted and proved in that module. From the work in this dissertation, it is clear that much of the structure of the extensions to first-order logic can be automatically inferred from program declarations.

- For each new data type declared, a set of theorems for typing, conversion, etc. can be automatically derived (see §4.3.2).
- For each new abstract data type declared, a new equality can be defined provided that the resulting congruences are proved (see §4.5.3).
- Certain properties of data types (such as admitting a test for equality) can be inferred and the corresponding programs constructed. For example, the type $\mathsf{List}(A)$ admits a test for equality provided the type A does. Formalised in CCL, this is the following theorem.

$$\frac{\mathsf{eq}_A : EQTYPE(A)}{\mathit{listeq}(\mathsf{eq}_A) : EQTYPE(\mathsf{List}(A))}$$

  in which $\mathit{listeq}(\mathsf{eq}_A)$ is a program that determines if two lists are equal provided that eq_A determines equality on their elements. $EQTYPE$ is defined in §4.5.2.

- For each theorem $a : A$, a set of conversions can be derived for the program a.
- Modules structure both code and theory; their signatures introduce new constants (program identifiers and types) and axioms. Interpreting one module in another requires theorem proving and is, therefore, interactive. Modules may be introduced to develop theory without appearing in programs; for example, the use of finite set theory in the development of unification.

Correctness assertions are proved in logical theories derived from the context in which they are made. Type checking in CCL generates correctness conditions from specifications, but the majority of any interactive reasoning is in the inferred extensions to first-order logic. In large programs, proving the correctness of the entire program is not always feasible; it is essential that correctness can be limited to parts of the code and assumptions made about the behaviour of other parts. In such circumstances, an environment for correctness must make explicit when axioms are introduced that are not discharged by underlying theories.

Appendix A

Well-Founded Induction

This appendix introduces *well-founded relations*, the principle of *well-founded induction*, and constructions over relations that preserve well-foundedness. In particular, the use of well-founded induction to support reasoning about general recursion is considered, subsuming amongst others the notions of structural induction and course-of-values induction.

§A.1 presents several equivalent definitions of well-foundedness, one of which gives rise to the principle of well-founded induction. §A.2 introduces constructions over relations that preserve well-foundedness.

A.1 Well-Founded Relations and Induction

A relation $\prec$ is *well-founded* over the set A if there exist no infinite descending chains of elements in A.

$$\ldots \prec x_n \prec \ldots \prec x_3 \prec x_2 \prec x_1$$

Rather than using infinite chains, well-foundedness can be defined in terms of *minimal* element subsets. For the set A, an element a is $\prec$-minimal iff there is no element $x \in A$ for which $x \prec a$.

Definition A.1 (Well-foundedness I) *A relation $\prec$ is well-founded over the set A iff every non-empty subset of A has an $\prec$-minimal element.*

Assuming dependent choice, this definition is equivalent to the informal statement above since

- $\prec$-minimal subsets $\Rightarrow$ no infinite descending chains
  Suppose there exists an infinite descending chain $\ldots \prec x_3 \prec x_2 \prec x_1$ in A. The set of elements in this chain $X = \{x_1, x_2, \ldots\}$ has an $\prec$-minimal element, since $X \subseteq A$; but for every element x_n in the chain, there exists an element x_{n+1} such that $x_{n+1} \prec x_n$—contradiction.

- no infinite descending chains $\Rightarrow$ $\prec$-minimal subsets
  Suppose there exists a non-empty set $X \subseteq A$ for which there is no $\prec$-minimal element. For an element $x_1 \in X$, there must exist another element $x_2 \in X$ such that $x_2 \prec x_1$. Repeating this argument for x_2 gives x_3 etc., generating an infinite descending chain—contradiction.

A well-founded relation cannot be reflexive (as this would permit the infinite chain $\ldots \prec x \prec x \prec x \prec x$) or symmetric ($\ldots \prec y \prec x \prec y \prec x$), though it may be transitive (Theorem A.3). For example, the predecessor relation is well-founded over natural numbers,

$$x \prec_{\mathrm{N}} y \quad \leftrightarrow \quad y = x + 1$$

and the immediate sublist relation is well-founded over lists.

$$x \prec_{\mathrm{L}} y \quad \leftrightarrow \quad y = (\mathit{cons}\ h\ x) \qquad \mathit{for\ some} \quad h$$

There is an alternative formulation of well-foundedness, which leads to a rule of well-founded induction.

Definition A.2 (Well-foundedness II) *A relation $\prec$ is well-founded over the set A iff for every unary predicate P*

$$\Big(\ \forall x \in A.\ (\forall y \in A.\ y \prec x \supset P(y)) \supset P(x)\ \Big) \supset \forall x \in A.\ P(x)$$

The outermost implication in the above definition is often written as a bi-implication; as the missing implication is a tautology, its inclusion would add nothing. This definition is the only one that is intuitionistically meaningful. In a classical setting, it is equivalent to the first definition.

Theorem A.1 *Definitions I and II, of well-foundedness, are equivalent.*

Proof: For an arbitrary relation $\prec$ and set A, show that $\prec$ satisfies Definition I iff it satisfies Definition II.

Definition I holds iff for $X = \{x \in A, \neg P(x)\}$, where P is arbitrary, if set X is non-empty then it contains an $\prec$-minimal element. This is formally expressed as

$$(\exists x \in A.\ \neg P(x)) \supset \exists x \in \{x \in A, \neg P(x)\}.\ \forall y \in \{x \in A, \neg P(x)\}.\ \neg y \prec x$$

which by simplification is equivalent to the formula

$$(1) \qquad (\exists x \in A.\ \neg P(x)) \supset \exists x \in A.\ \neg P(x) \wedge \forall y \in A.\ y \prec x \supset P(y)$$

Taking the contrapositive form of (1) and again simplifying yields

$$\Big(\ \forall x \in A.\ (\forall y \in A.\ y \prec x \supset P(y)) \supset P(x)\ \Big) \supset \forall x \in A.\ P(x)$$

which is just definition II. ■

Definition II leads to a principle of well-founded induction. Writing $\mathit{Wfd}_A(\prec)$ for the assertion that the relation $\prec$ is well-founded over set A, this is expressed by the following theorem.

Theorem A.2 (Well-founded induction) *If*

i. $\mathit{Wfd}_A(\prec)$,

ii. $\forall x \in A.\ (\forall y \in A.\ y \prec x \supset P(y)) \supset P(x)$

then $\forall a \in A.\ P(a)$.

Proof: Trivially from the second definition of well-foundedness. ■

Expressed as a rule of natural deduction, this theorem has the following form (used in later proofs).

$$\frac{Wfd_A(\prec) \qquad a \in A \qquad \begin{matrix} \left[\; x \in A \;;\; \forall y{\in}A.\; y \prec x \;\supset\; P(y) \;\right]_x \\ P(x) \end{matrix}}{P(a)}$$

Well-founded induction generalises the more familiar notions of structural induction and course-of-values induction. For example, structural induction over the natural numbers, more familiarly known as mathematical induction, corresponds to well-founded induction with the predecessor relation; course-of-values induction on natural numbers corresponds to well-founded induction with the less-than relation, which is the transitive closure of the predecessor relation.

A.2 Constructing Well-Founded Relations

Often the easiest way to prove that a relation is well-founded is by construction. The following constructions for relations are shown to preserve well-foundedness: the irreflexive transitive closure of a relation, the mapping of one relation on to another, and the lexicographic ordering of two relations. These constructions are sufficient for proving termination in many examples. One exception is an algorithm for proof normalisation in which recursion follows a multi-set ordering (though an additional constructor could be added for multi-set orderings).

A.2.1 Irreflexive Transitive Closure

The irreflexive transitive closure of a relation $\prec$, written $\prec^+$, is the least relation such that

$$a \prec^+ b \quad \textit{iff} \quad \Big(a \prec b \;\vee\; (\exists x. a \prec^+ x \wedge x \prec b) \Big) \;\wedge\; a \neq b$$

Theorem A.3 *Let $\prec$ be a well-founded relation over the set A, then $\prec^+$ is also well-founded over A.*

Proof: By the second definition of well-foundedness, assume

(1) $$\forall x{\in}A.\; (\forall y{\in}A.\; y \prec^+ x \;\supset\; P(y)) \;\supset\; P(x)$$

and prove $\forall a{\in}A.\; P(a)$. By (1), this holds if the following holds

(*) $$\forall b{\in}A.\; b \prec^+ a \;\supset\; P(b)$$

By well-founded induction, using relation $\prec$, (*) holds if assuming

(2) $$\forall x{\in}A.\; x \prec c \;\supset\; \forall y{\in}A.\; y \prec^+ x \;\supset\; P(y)$$

the formula $\forall b{\in}A.\; b \prec^+ c \;\supset\; P(b)$ holds. By the definition of $\prec^+$, the hypothesis $b \prec^+ c$ can be rewritten to a disjunction, so that this holds if the following two formulae hold

(†) $b \prec c \;\supset\; P(b)$

(‡) $(\exists x{\in}A.\; b \prec^+ x \;\wedge\; x \prec c) \;\supset\; P(b)$

For (†), the hypothesis and (2) imply $\forall\, y \in A.\ y \prec^+ b \supset P(y)$, which by (1) implies $P(b)$.
For (‡), the hypothesis and (2) immediately imply $P(b)$. ■

From this result, it follows that less-than $<$ (i.e. $\prec_{\mathrm{N}}^+$) is well-founded over the natural numbers, and that the strict sublist relation $\prec_{\mathrm{L}}^+$ is well-founded over lists.

A.2.2 Mapping

Theorem A.4 *Let $\prec_B$ be a well-founded relation over the set B and $f \in A \to B$ a total function. Then any relation $\prec_A$ over A for which*

$$x \prec_A y \ \supset\ f(x) \prec_B f(y) \qquad \textit{for all} \quad x, y \in A$$

is well-founded over A.

Proof: By the second definition of well-foundedness, assume

(1) $$\forall\, x \in A.\ (\forall\, y \in A.\ y \prec_A x \supset P(y)) \supset P(x)$$

and prove $\forall\, a \in A.\ P(a)$. This can be strengthened to

(*) $$\forall\, b \in B.\ \forall\, a \in A.\ f(a) = b \supset P(a)$$

which clearly implies $\forall\, a \in A.\ P(a)$ in the case $b = f(a)$.
By well-founded induction, using relation $\prec_B$, (*) holds if assuming

(2) $$\forall\, y \in B.\ y \prec b \supset \forall\, a \in A.\ f(a) = y \supset P(a)$$

the formula $\forall\, a \in A.\ f(a) = b \supset P(a)$ holds.
By (1) and the hypothesis, $f(a) = b$, this holds if assuming

(3) $$f(a) = b$$

(4) $$y \prec_A a$$

the formula $P(y)$ holds.
By the assumed property of $\prec_A$, (4) implies $f(y) \prec_B f(a)$. Rewriting $f(a)$, using (3), gives $f(y) \prec_B b$, which can be used in assumption (2) to give

(5) $$\forall\, a \in A.\ f(a) = f(y) \supset P(a)$$

For $a = y$, this immediately proves $P(y)$. ■

As corollaries of the mapping theorem, the following results hold for induced relations, subrelations and subsets.

Theorem A.5 *Let $\prec_B$ be a well-founded relation over the set B and $f \in A \to B$ a total function. Then the relation $\prec_A$ over A defined by*

$$x \prec_A y \quad \textit{iff} \quad f(x) \prec_B f(y) \qquad \textit{for all} \quad x, y \in A$$

is well-founded over A.

Theorem A.6 *Let $\prec$ be a well-founded relation over the set A and $\prec'$ a relation over A for which $\prec' \subseteq \prec$, then $\prec'$ is well-founded over A.*

Theorem A.7 *Let $\prec$ be a well-founded relation over the set A and $B \subseteq A$. Then $\prec$ is well-founded over B.*

The following examples show how these results allow new relations to be constructed.

- From the mapping theorem, it follows that the strict subset relation is shown to be well-founded by the following property.

  $$A \subsetneq B \supset \mathit{card}(A) < \mathit{card}(B)$$

  that is if A is a strict subset of B then the cardinality of A is less-than that of B.

- From the induced relation theorem, it follows that the relation induced by the length function for lists (*length*) is well-founded.

  $$k \prec_{\text{len}} l \leftrightarrow \mathit{length}(k) < \mathit{length}(l)$$

- From the subrelation theorem, it follows that the predecessor relation is well-founded over natural numbers since the relation $<$ is well-founded over natural numbers.

- From the subset theorem, it follows that the relation $<$ is well-founded over natural evens since it is well-founded over the natural numbers.

A.2.3 Lexicographic Ordering

Theorem A.8 *Let $\prec_A$ and $\prec_B$ be well-founded relations over the sets A and B respectively. Define a new relation $\prec$, the lexicographic ordering of these, as*

$$\langle x, y\rangle \prec \langle x', y'\rangle \quad \mathit{iff} \quad x \prec_A x' \lor x = x' \land y \prec_A y'$$

then $\prec$ is well-founded over $A \times B$.

Proof: By the second definition of well-foundedness, assume

(1) $$\forall p \in A \times B.\ (\forall q \in A \times B.\ q \prec p \supset P(q)) \supset P(p)$$

and prove $\forall r \in A \times B.\ P(r)$, that is

(*) $$\forall a \in A.\ \forall b \in B.\ P(\langle a, b\rangle)$$

By well-founded induction, using relation $\prec_A$, (*) holds if assuming

(2) $$\forall x \in A.\ x \prec_A a \supset \forall b \in B.\ P(\langle x, b\rangle)$$

the formula $\forall b \in B.\ P(\langle a, b\rangle)$ holds. By well-founded induction, using relation $\prec_B$, this holds if assuming

(3) $$\forall y \in B.\ y \prec_B b \supset P(\langle a, y\rangle)$$

the formula $P(\langle a, b\rangle)$ holds. By (1), this holds if assuming

(4) $$\langle u, v\rangle \prec \langle a, b\rangle$$

the formula $P(\langle u, v\rangle)$ holds. By the definition of $\prec$, (4) can be rewritten as a disjunction, and $P(\langle u, v\rangle)$ holds if the following two formulae hold

(†) $\quad u \prec_A a \supset P(\langle u, v\rangle)$
(‡) $\quad u = a \;\wedge\; v \prec_B b \;\supset\; P(\langle u, v\rangle)$

Now, (†) immediately follows from (2) and, after substituting $u = a$, (‡) follows from (3). ■

The result for lexicographic pairing can be extended to all finite tuples, by considering them as nested pairs. But the usual dictionary ordering over strings (i.e. lists of characters) does not fall into this category. The dictionary ordering is a lexicographic style of ordering extended to lists, rather than tuples of fixed length; it is not well-founded as the following example illustrates.

$$\ldots aaab \prec aab \prec ab \prec b$$

But ordering strings first by their length and then lexicographically is well-founded.

References

[1] Samson Abramsky. The lazy lambda calculus. In David Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.

[2] Krzysztof R. Apt. Ten years of Hoare's logic: A survey—Part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, 1981.

[3] R. Backhouse, P. Chisholm, G. Malcolm, and E. Saaman. Do-it-yourself type theory. *Formal Aspects of Computing*, 1(1):19–84, 1989.

[4] J. L. Bates and R. L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, 1985.

[5] R. S. Bird. Formal derivation of a pattern matching algorithm. *Science of Computer Programming*, 12:93–104, 1989.

[6] R. S. Bird. A calculus of functions for program derivation. In David Turner, editor, *Research Topics in Functional Programming*, pages 287–307. Addison-Wesley, 1990.

[7] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[8] P. Chisholm. Derivation of a parsing algorithm in Martin-Löf's theory of types. *Science of Computer Programming*, 8:1–42, 1987.

[9] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[10] Avra Cohn and Robin Milner. On using Edinburgh LCF to prove the correctness of a parsing algorithm. Technical Report CSR-113-82, Department of Computer Science, University of Edinburgh, May 1982.

[11] Avra J. Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning*, 5(2):127–139, 1989.

[12] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall International, 1986.

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

[14] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Symposium on Principles of Programming Languages*, pages 207–212. ACM Press, 1982.

[15] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 1990.

[16] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall International, 1976.

[17] Peter Dybjer. Program verification in a logical theory of constructions. In *Functional Programming Languages and Computer Architecture*, pages 334–349. Springer-Verlag, 1985. LNCS 201.

[18] Lars-Henrik Eriksson. Synthesis of a unification algorithm in a logic programming calculus. *Journal of Logic Programming*, 1(1):3–18, 1984.

[19] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proof and Types.* Cambridge University Press, 1989.

[20] Michael J. C. Gordon. HOL: A proof generating system for higher-order logic. In Graham Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, 1988.

[21] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation.* Springer-Verlag, 1979. LNCS 78.

[22] D. Gries. *The Science of Programming.* Texts and Monographs in Computer Science. Springer-Verlag, 1981.

[23] Carl A. Gunter. Forms of semantic specification. *Bulletin of EATCS*, 45:98–113, 1991.

[24] Susumu Hayashi and Hiroshi Nakano. *PX : A Computational Logic.* Foundations of Computer Science. MIT Press, 1989.

[25] Martin C. Henson. Program development in the constructive set theory TK. *Formal Aspects of Computing*, 1:173–192, 1989.

[26] Arend Heyting. *Intuitionism: An Introduction.* North-Holland, 1956.

[27] R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[28] Douglas J. Howe. Equality in lazy computation systems. In *Symposium on Logic in Computer Science*, pages 198–203. IEEE Computer Society Press, 1989.

[29] Paul Hudak, Simon Peyton Jones, and Philip Wadler. Report on the programming language Haskell, version 1.1, August 1991.

[30] G. P. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[31] G. Kahn. Natural semantics. In F. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer-Verlag, 1987. LNCS 247.

[32] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, January 1964.

[33] Daniel Leivant. Contracting proofs to programs. In P. Odifreddi, editor, *Logic and Computer Science*, pages 279–327. Academic Press, 1990.

[34] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[35] Z. Manna and R. Waldinger. Deductive synthesis of the unification algorithm. *Science of Computer Programming*, 1:5–48, 1981.

[36] Z. Manna and R. Waldinger. The origins of a binary search paradigm. *Science of Computer Programming*, 9(1):37–83, 1987.

[37] Z. Manna and R. Waldinger. *The Logical Basis for Computer Programming: Deductive Systems*. Addison-Wesley, 1990.

[38] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[39] Per Martin-Löf. Intuitionistic type theory. Bibliopolis, Napoli, 1984.

[40] Per Martin-Löf. Constructive mathematics and computer programming. In C. A. R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 167–184. Prentice-Hall International, 1985.

[41] J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, 1963.

[42] Thomas F. Melham. Automating recursive type definitions in higher order logic. In Graham Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386. Springer-Verlag, 1989.

[43] Albert R. Meyer. Semantical paradigms: Notes for an invited lecture. In *Symposium on Logic in Computer Science*, pages 236–253. IEEE Computer Society Press, 1988.

[44] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[45] Christine Mohring. Algorithmic development in the calculus of constructions. In *Symposium on Logic in Computer Science*, pages 84–91. IEEE Computer Society Press, 1986.

[46] Tobias Nipkow. Order-sorted polymorphism in Isabelle. In G. Huet, G. Plotkin, and C. Jones, editors, *Proc. 2nd Workshop on Logical Frameworks*, pages 307–321, 1991.

[47] B. Nordström. Programming in constructive set theory: Some examples. In *Functional Programming Languages and Computer Architecture*, pages 141–153. ACM Press, 1981.

[48] B. Nordström. Terminating general recursion. *BIT*, 28:605–619, 1988.

[49] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press, 1990.

[50] Christine Paulin-Mohring. Extracting F_ω's programs from proofs in the calculus of constructions. In *Symposium on Principles of Programming Languages*, pages 89–104. ACM Press, 1989.

[51] Lawrence C. Paulson. Verifying the unification algorithm in LCF. *Science of Computer Programming*, 5:143–169, 1985.

[52] Lawrence C. Paulson. Constructing recursion operators in intuitionistic type theory. *Journal of Symbolic Computation*, 2:325–355, 1986.

[53] Lawrence C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF.* Cambridge Tracts in Theoretical Computer Science 2. Cambridge University Press, 1987.

[54] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[55] Lawrence C. Paulson and Tobias Nipkow. Isabelle tutorial and user's manual. Technical Report 189, University of Cambridge Computer Laboratory, 1990.

[56] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report FN–19, Computer Science Department, Aarhus University, September 1981.

[57] D. Prawitz. *Natural Deduction: A Proof-Theoretical Study.* Almquist and Wiksell, 1965.

[58] J. A. Robinson. Computational logic: The unification computation. *Machine Intelligence*, 6:63–72, 1971.

[59] Anne Salvesen and Jan M. Smith. The strength of the subset type in Martin-Löf's type theory. In *Symposium on Logic in Computer Science*, pages 384–391. IEEE Computer Society Press, 1988.

[60] Donald Sannella and Andrzej Tarlecki. Toward formal development of ML programs: Foundation and methodology. Technical Report ECS-LFCS-89-71, Department of Computer Science, University of Edinburgh, February 1989.

[61] P. Schroeder-Heister. A natural extension of natural deduction. *Journal of Symbolic Logic*, 49(4):1284–1300, December 1984.

[62] Jan M. Smith. On a nonconstructive type theory and program derivation. In *Conference on Logic and its Applications.* Plenum, 1986. Bulgaria.

[63] J. Traugott. Deductive synthesis of sorting programs. In J. Siekmann, editor, *International Conference on Automated Deduction*, pages 641–660. Springer-Verlag, 1986.

An Index to Notation

		Introduced on page
‘	Call-by-name application in $\mathcal{L}$.	23
ˆ	Call-by-value application in $\mathcal{L}$.	23
$+$	A defined program type in CCL.	50
$\times$	A defined program type in CCL.	29
$:$	The type membership relation in CCL.	30
$\{\}$	A constructor for program types in CCL.	29
$\langle\rangle$	A constructor of $\mathcal{L}$.	21
$=$	The equality relation on terms (and on types) in CCL.	26 (30)
$\downarrow$	The termination predicate for programs of $\mathcal{L}$.	24
$\rightarrow$	A defined program type in CCL.	29
$\rightsquigarrow$	The reduction relation for $\mathcal{L}$.	21
$\triangleright$	The evaluation relation for $\mathcal{L}$.	22
$\mathcal{L}$	The target programming language.	21
ι	The set of program terms of $\mathcal{L}$.	21
$\iota_=$	The set of equivalence classes of programs (ι) under term equality (=)—a meta-type in the Isabelle implementation.	27
τ	The set of program types—a meta-type in the Isabelle implementation.	29
ω	The set of well-founded orderings—a meta-type in the Isabelle implementation.	34
o	Logical formulae of CCL—a meta-type in the Isabelle implementation.	19
μ	A constructor for program types in CCL.	29
$\prec_R$	The mebership relation on well-founded orderings in CCL.	19
Π	A constructor for program types in CCL.	29
Σ	A constructor for program types in CCL.	29

An Index to Notation contd.