

This report is a revised version of

Computer Science Technical Report Number CS-081,
Dept. of Electrical Engineering and Computer
Sciences, Univ. of Calif., San Diego, U.S.A.,

June 1984

June 1984 (Revised February 1985)

An Approach to Program Reasoning Based on a First Order Logic for Partial Functions.

Olaf Owe

Institute of Informatics
University of Oslo
Norway

Abstract

Formal program reasoning must be based on some underlying logic. Because program reasoning requires partial functions, traditional first order logic is not satisfactory. There is a need for a logic that allows partial functions, and there is a need for a concept of definedness that corresponds to that of normal termination of programs. In order to meet these needs, several formal systems for reasoning about partial functions have already been suggested.

This paper states three requirements for such a formal system and argues that these requirements are essential in order to allow a realistic abstraction concept which permits "implementations errors" such as capacity constraints. The purpose of this paper is to present a first order logic for partial functions that satisfies these requirements, and, in addition, is consistent and complete.

The presented logic is called Weak Logic because it has a "weak" semantics of formulas, as well as a "weak" semantics of universal quantifiers: A formula is valid if it is true whenever it is defined. The meaning of $\forall x : T [q]$ is that q is true for every value x in T such that q is defined. However, Weak Logic has a "strong" semantics of existential quantifiers. The meaning of $\exists x : T [q]$ is that there is a value x in T such that q is both defined and true. By an explicitly defined definedness operator, Weak Logic enables us to express, and reason about, definedness properties of formulas and expressions.

Categories and Subject Descriptions:

F.3.1 [Logics and Meaning of Programs]: Specifying and Verifying and Reasoning about Programs.

F.3.2 [Logics and Meaning of Programs]: Semantics of Programming Languages.

F.4.1 [Mathematical Logic and Formal Languages]: Mathematical Logic.

General Terms: Axiomatic Semantics, First Order Logic, Partial Functions, Definedness

Additional Key Words and Phrases: Implementation Errors, Exception Handling, Partial Implementations, Abstract Data Types

Table of Contents

Page

1	Introduction.....	1
4	1 Why Weak Semantics?.....	4
6	2 Formalization of Definedness.....	6
8	3 Base Logic.....	8
10	2 Weak Logic.....	10
12	1 Introduction of User-defined Functions and Types.....	12
16	2 Expression Language.....	16
17	2.1 Non-kernel Expressions.....	17
18	3 The Definedness Operator.....	18
20	4 Base Logic.....	20
23	4.2 Extended Base Logic.....	23
23	5 Validity.....	23
25	6 Provability.....	25
28	3 Main Properties of Weak Logic.....	28
28	1 Consistency and Completeness.....	28
36	2 Relationship to Base Logic.....	36
36	3 Approximations.....	36
38	3.1 Approximation of Functions.....	38
39	3.2 Approximation of Types.....	39
41	4 Possible Extensions.....	41
41	1 Exception Handling.....	41
43	Appendix A.....	43
44	References.....	44
45	Acknowledgements.....	45

1. Introduction

The relationship between in- and out- values of a subprogram is commonly characterized by a function, called an effect function. Since a subprogram may terminate exceptionally or may not terminate at all, the effect function is a partial function, defined whenever the subprogram terminates normally, and undefined otherwise. This implies that there is a need for partial functions in program reasoning, and that there is a need for a concept of definedness that corresponds to that of normal termination of programs. Clearly, traditional first order logic [Sch 67] is not satisfactory because it requires total (i.e., universally defined) functions. In order to reason about partial functions, we need a logic that can give meaning to formulas that are undefined (i.e., do not have a value). Several suggestions for such formal systems have already been given, for instance [Bar 84, Gor 79, Kir 82, Luc 84]. Rather than aiming at a system with nice mathematical properties, such as the "rule of not-elimination" below,

$$\frac{\text{--- } q, \text{ --- not } q}{\text{--- False}}$$

(which says that if the formulas q and $\text{not } q$ are theorems, then False is a theorem), we shall concentrate on properties that are important in the context of program reasoning. Motivated by the importance of the concept of abstraction (both data abstraction and functional abstraction) [Dah 78], we have found three major requirements:

- Req. 1. Reasoning about defined formulas should follow the rules of traditional first order logic, so that reasoning about normally terminating programs can be done in the usual manner.
- Req. 2. If a theorem q is proved on an abstract level, q should be valid on a less abstract level as well. At least, this rule should apply to a reasonable set of possible q 's; otherwise, the concept of abstraction would be useless in program reasoning.
- Req. 3. On an abstract level, one should not be forced to specify, or identify, so-called implementation dependent errors (as described below). This implies that an expression which is defined on an abstract level can be undefined on a less abstract level. (However, abstract errors, such as 1/0 or *pop (empty)*, should not have a defined value on a less abstract level.)

Obviously, Requirement 1 discourages three valued logic. Requirements 2 and 3 may seem too strong: For instance, let, according to Requirement 3, e be an expression which is abstractly defined, but undefined on a less abstract level. Thus the statement " e is defined" should be provable on the abstract level, whereas the statement " e is defined" is obviously not valid on the less abstract level; and hence Requirement 2 is violated. Therefore, we must either disallow certain operations or restrict their use (as indicated in Requirement 3) -- in fact, certain "non-strict" operators, such as "is defined" and non-strict equality, are the problematic ones.

Our goal is to design a formal system that satisfies the three requirements above. In order to be useful, it should also be consistent and complete, and its proof rules and axioms should be as simple as possible, and finally, it should be possible to introduce user-defined functions (and predicates), as well as user-defined axioms to characterize these functions.

The three requirements above were first mentioned in [Owe 80]. The present work is a revised and simplified version of [Owe 82], and is based on ideas from [Owe 80]. No other attempt to design a formal system that satisfies the three requirements above is known to the author.

As described in Section 1.1, we argue that our three requirements motivate a "weak" interpretation of validity:

A formula is considered valid if it is true whenever it is defined.

It also motivates a "weak" interpretation of universal quantifiers and a "strong" interpretation of existential quantifiers:

$\forall x : T. q$ is valid if q is true for all values x in T such that q is defined.
 $\exists x : T. q$ is valid if there is a value x in T such that q is defined and is true.

Based on these ideas, we introduce, in section 2, a first order logic allowing partial functions, called Weak Logic, which satisfies our three requirements. We present a consistent and complete (in the sense that provability is equivalent to validity) set of proof rules for Weak Logic, consisting of four basic proof rules, including Modus Ponens, which is the only basic rule that differs from the traditional ones. Weak Logic does not satisfy the rule of not-elimination above; however, it does satisfy the following (derived) not-elimination rule:

$$\frac{\vdash q, \vdash \text{not } q}{\vdash \text{not } DEF[q]}$$

where DEF is the definedness operator of Weak Logic (see section 2.3).

For a formula q , the question "is q defined?" may often be harder to prove than the question "is q true, provided q is defined?" Within Weak Logic, it is possible to address and prove these questions independently. In fact, one can prove that q is defined and true by proving (i) and (ii) below:

- (i) $\vdash q$ (which intuitively means that q is true whenever defined)
- (ii) $\vdash DEF[q]$ (which denotes a formula expressing "is q defined?")

Note that (i) can very well be proved before (ii), and that both proofs can be done within Weak Logic.

In order to formally handle the concept of implementation dependent errors, we introduce a generalized concept of abstraction, called approximation (similar to idealization in [Owe 80]), denoted by the symbol $\sqsubseteq$. The intuitive meaning of $A \sqsubseteq B$ is that A partially (non-strictly) implements B ; for instance, a partial implementation of a function f may be less defined than f , but when defined, it may not conflict with f . Similarly, a partial implementation of a type T will satisfy the basic properties of

T' , but its associated functions may be less defined than those of T , and its set of values may be more constrained than that of T . For instance, *finite stack* $\equiv$ *infinite stack*, and bounded integer (with partial implementations of the usual integer operations) $\equiv$ (unbounded) integer. Note that the word partial refers to definedness, and not to completeness (of user-defined axiom systems).

In order to fulfill Requirement 2 above, one must prove that concepts (types and functions) on a less abstract level approximate (partially implement) those on a more abstract level. Requirement 2 is fulfilled by the following "Rule of Approximation":

$$\frac{T' \vdash q}{T \vdash q}$$

provided $T' \equiv T$, and where q' is q with each T' -function replaced by the corresponding T -function (equality on T' may correspond to an equivalence relation on T'), and where the formula q may not contain any existential quantifiers (in the prenex form). (For more details see section 3.3.) Since such an existential quantifier can be removed by introducing an auxiliary function (Gut 75), and since typical data type axioms do not have existential quantifiers, this restriction seems reasonable.

Finally, as another example of the usefulness of Weak Logic we show that Hoare Logic [Hoa 69],

when based on Weak Logic, rather than traditional first order logic, becomes (relatively) complete. Some of the traditional proof rules of Hoare Logic need to be slightly adjusted, but no additional proof rule to handle undefinedness is needed: for instance, the assignment rule is the same. In the modified Hoare Logic, the intuitive meaning of the notation $p \{ S \} q$ is: if the formula p is true or undefined in the initial state of the program S , and if S terminates normally, then the formula q is true or undefined in the final state of S . As an example, we may prove

$$x = 0 \{ x := 1/x \} \text{false}$$

which is not provable in the traditional Hoare System. We also show that approximation is a useful concept in Hoare-like reasoning, as (informally) stated by the following rule:

$$\frac{p \{ S' \} q}{p \{ S \} q}$$

provided S approximates S' . This means that in order to reason about a program S , one may reason about an abstract program S' such that S approximates S' . Finally, we show how to generate an abstract program S' that terminates normally over a specific domain (given p , S , and q). These results are reported in [Owe 82 & Owe 85].

1.1. Why Weak Semantics

In program reasoning, it is useful to define several abstraction layers. For instance, the specification of a data type defines one abstraction level, and an implementation defines a less abstract level. On each level, a set of definitions and/or axioms is available (stating properties about types and functions), which, in turn, enable us to prove theorems. In program reasoning, results from one level are applied to less abstract levels. For instance, program reasoning is usually based on a mathematical axiomatization of integers even if our implementation only caters for bounded integers. This shows why Requirement 2 above is important. Now, how does this requirement affect the concept of definedness? Consider the formulas

$$(A1) \quad \vdash x \text{ integer} \rightarrow x < \text{succ}(x)$$

$$(A2) \quad \vdash s \text{ Set}, x \text{ Item} \mid \text{has}(\text{add}(s, x), y) = (x = y) \text{ or } \text{has}(s, y)$$

which are two examples of typical (abstract) axioms. An executable implementation can only define a finite set of values; for instance, there is a largest representable integer, say *Max*. This means that

succ (*Max*) is defined on the abstract level and undefined on the implementation level. Consider the formula

$$\text{Max} < \text{succ}(\text{Max})$$

This formula is undefined on the implementation level (since its evaluation would terminate by an exception), but it is defined and provable on the abstract level (as an obvious consequence of A1). In order to avoid such situations one could restate A1 and A2 as follows:

$$(A1') \quad \forall x \text{ integer} \mid \text{if } x < \text{Max} \text{ then } x < \text{succ}(x) \text{ else True}$$

$$\forall s \text{ Set}, x \text{ Item}$$

$$(A2') \quad \text{if not full}(s) \text{ then has}(\text{add}(s, x), y) = (x = y) \text{ or has}(s, y) \text{ else True}$$

Is this a satisfactory solution? First, note that, since different implementations have different values of

Max, it is impossible to state any abstract properties of *Max*. Considering the set example, one

"remove-efficient" implementation may sort and store all distinct items added, whereas another "add-efficient" implementation may store all items (distinct or not) in the order added into the set. With the latter implementation, the result of the full-predicate can be different for abstractly equal sets. This means that it would be impossible to give a complete (abstract) axiomatization of the set type, whereas with axioms of the form A2, completeness can be achieved. In addition, axioms of form A1' and A2' would make it quite awkward to prove anything, since there would be little abstract knowledge about constants and functions like *Max* and *full*.

We conclude that axioms of form A1' and A2' are not satisfactory; and further, that on an abstract level, it is not desirable to identify (or specify) "implementation dependent" error situations, such as

$suc(Mar)$, and $add(s, x)$ when s is full. This shows why Requirement 3 is important.

Now, since we really do want axioms like A1 and A2 on the abstract level, formulas like

$$Mar < suc(Mar)$$

must be valid on the implementation level according to Requirement 2. It may even be that

$$not\ Mar < suc(Mar)$$

is provable if $\forall x : Integer [x \leq Mar]$ is a theorem on the implementation level. The only possible

solution seems to be the following (informal) interpretation of formulas:

A formula is valid if it is true whenever it is defined.

Hence an undefined formula, such as $Mar < suc(Mar)$, as well as $not\ Mar < suc(Mar)$, is trivially

valid.

Secondly, the axioms A1 and A2 themselves must be valid on the implementation level. This sug-

gests the following weak interpretation of universal quantifiers:

$\forall x : T, q$ is valid if q is true for all values x in type T such that q is defined.

This interpretation allows axioms (or theorems) like

$$\forall x : Integer [x / r = 1]$$

$$\forall s : Stack [length(pop(s)) = length(s) - 1]$$

even if y / x is undefined when x is 0, and $pop(s)$ is undefined when s is empty. The traditional intro-

duction of existential quantifiers:

$$\exists x : T [q] = not\ \forall x : T [not\ q]$$

implies the following (strong) interpretation of existential quantifiers:

$\exists x : T [q]$ is valid if there is a value of x in T such that q is defined and true.

This interpretation seems reasonable. For instance, for a typical Stack axiomatization (see section 3.3),

$$\exists s : Stack [pop(s) = empty]$$

would be valid, but not

$$\exists s : Stack [s = pop(empty)]$$

With a strict (i.e., undefinedness preserving) interpretation of existential quantifiers, both formulas above

would be undefined (and therefore trivially valid). With a "weak" interpretation of existential quantifiers,

both formulas above would be valid. Neither of the two alternative interpretations is appealing.

An axiom of form $\forall x_1 : T_1 [\dots \forall x_n : T_n [s = rs] [\dots]]$ can be interpreted as a rewrite rule. Rewrite

rule systems based on first order logic are discussed in [Knu 70]. In order to apply these results to Weak

Logic, we must add the following requirement to each rewrite rule: The righthand side (rs) must be defined whenever the lefthand side (ls) is defined. This will ensure that if a formula p can be rewritten as $True$ ($False$) by a set of rewrite rules, we can conclude that p ($not\ p$) is valid. (For more details, see [Owe 80].)

1.2. Formalization of Definedness

As already stated, we need a first order logic with a concept of definedness that corresponds to that of normal termination, and where undefined (and partly defined) formulas have a meaning. Moreover, the meaning of such formulas should be as explained in section 1.1 above. This section will describe how it is possible to express, axiomatize, and reason about definedness properties of functions, types, and expressions. Let us first introduce some terminology.

An expression is said to be defined if it is always defined (i.e., for all values of the free variables). (A formal definition is given in section 2.3.) Similarly, an expression is said to be undefined if it is always undefined; and it is said to be partly defined if it is neither undefined nor defined.

A boolean function is called a predicate, and a boolean expression is called a formula. A function with an empty domain is called a constant. A function is said to be strict if an undefined argument yields an undefined function value. Thus, a non-strict function is defined for (at least) some undefined arguments. For instance, the if-construct is non-strict according to the usual execution rules, since an if-expression such as

if $x = 0$ then 0 else 1 / x fi

has a defined value even when the subexpression 1 / x is undefined. A function is said to be strictly total if it is strict and defined for all defined arguments. For instance, equality (denoted =) and the abstract integer operations +, -, *suc*, and * are strictly total.

As mentioned, the effect of a subprogram is formalized by means of a function from the in-values (of the in and in/out parameters of the subprogram) to the out-values (of the in/out and out parameters, including returned function values, if any). Thus the effect function of a subprogram is strict if the subprogram terminates normally only when the evaluation of the in-values terminates normally. This is, in fact, true if the in-values are evaluated before the execution of the body of the subprogram, which is the case for the normal parameter passing modes such as call by value and/or result, call by constant, call by reference (provided referenced variables have a defined value), etc., but not for parameter passing modes such as call by name or call by lazy evaluation. Because reasoning about definedness (and type checking) of program variables is different for different programming languages, (for instance, in ABEL [Dah 85], program variables are always defined), we assume that all variables in Weak Logic are defined, which implies that normal termination of call by reference and call by value (etc.) is the same.

We restrict ourselves to the "normal" parameter passing rules, and consequently, all effect functions are strict. Since a user-defined function may describe a (possibly abstract) effect function of a subprogram, we therefore decide that all user-defined functions in our logic should be strict. Pre-defined functions should follow the usual execution rules, which means that equality and the boolean operations

and, *or*, *implies*, *not*, should all be strictly total (and therefore strict), whereas the if-construct (and the abbreviated if-forms: *and then*, *or else*, *implies then*) should be non-strict and non-commutative. As already mentioned, quantifiers should be non-strict, and in addition (in section 2.2), we introduce non-strict versions of the boolean operators: "and", "or", and "implies", denoted $\wedge$, $\vee$, and $\Rightarrow$, respectively. These non-strict boolean operators are non-executional, but will be convenient within formal reasoning and specification. For instance, if q is undefined, the formulas

$$(False \text{ and } q), (q \text{ and } False), (q \text{ and then } False)$$

are all undefined and therefore trivially valid, whereas the formulas

$$(False \text{ and then } q), (False \vee q), (q \wedge False)$$

are all defined and are invalid. The boolean operations *and*, *or*, as well as $\vee$ and $\wedge$ are commutative. A user-defined partial function f may not be defined for all arguments in its domain. In order to be able to reason about definedness, we need semantic knowledge about when f is defined. For this purpose, we shall associate to f a strictly total function, called the domain predicate of f , denoted in f :

$$x \text{ in } f \text{ is true if } f(x) \text{ is defined, and false otherwise.}$$

Just as with any user-defined function, the semantics of this predicate is more or less completely characterized through user-defined axioms and/or definitions (see section 2.1). (A similar discussion applies to semantically constrained types.) Domain predicates enable us to express and reason about definedness properties without requiring a complete specification of the definedness of partial functions. In fact, it is possible to express the definedness of an expression e by means of a formula, denoted $DEF[e]$, in terms of domain predicates and the free variables of e :

$$DEF[e] \text{ is true when } e \text{ is defined, and false otherwise.}$$

The formula denoted $DEF[e]$ is always defined. The DEF -operator is a (non-strict) meta function, which is explicitly defined in section 2.3.

From the discussion above, we may conclude that our first order logic, Weak Logic, must have the following characteristics:

- i) All user-defined functions, as well as equality and the boolean operators *and*, *or*, *implies*, and *not*, are strict.
- ii) The only basic non-strict constructs are if-constructs and quantifiers (and we will add non-strict boolean operators).

- iii) To each partial function f (and type symbol T), there is an associated domain predicate in f ($in\ T$). This predicate is total by definition, but may or may not be fully specified.
- iv) The concept of definedness is realized by the DEF -operator, which takes an expression into a defined formula.
- v) The semantics of defined formulas is the same as that of traditional first order logic. Undefined and partly defined formulas have a weak semantics, as explained in section 1.1 above.

1.3. Base Logic

In order to prove consistency and completeness of Weak Logic, we shall introduce a version of traditional first order logic with a trivial semantics of definedness. This simpler logic is called Base Logic.

(See section 2.4).

Weak Logic is consistent and complete if all valid formulas are provable, and if all provable formulas are valid: i.e.

$$p \quad \text{if and only if} \quad \vdash p$$

One way of proving this would be to define the validity concept of Weak Logic by means of Weak Logic models and interpretations of Weak Logic formulas into such models. Since this is already done for traditional first order logic, it suffices to define an interpretation from Weak Logic into traditional first order logic and then take advantage of the consistency and completeness results from traditional first order logic.

For this purpose, we shall use a version of traditional first order logic, called Base Logic. With this interpretation, the proof of completeness and consistency will become relatively simple, and the focus will be on the differences between Weak Logic and Base Logic; i.e., definedness issues. Hopefully, the proof itself will provide a good understanding and illustration of Weak Logic to the reader.

For reasons of simplicity, we would like to use the same expression language for both Base Logic and Weak Logic. We therefore allow partial functions in Base Logic, but use a trivial semantics of

definedness:

Only "strongly defined" formulas are accepted in Base Logic (see section 2.4), and the only basic non-strict construct is if-expressions. (In addition, non-strict versions of the boolean operations can be derived, just as in Weak Logic.) In particular, quantifiers are strict.

For instance, the formula

$$\forall s: Stack [length(pop(s)) = length(s) - 1]$$

is not acceptable in Base Logic, but the following formula is:

$$\forall s: Stack [length(s) > 0 \text{ implies then } length(pop(s)) = length(s) - 1]$$

As an interesting side result, we show that our formalization of Base Logic can easily be extended to allow undefined and partly defined formulas. The extended Logic is consistent and complete. In this

extended Base Logic, the intuitive meaning of $\models p$ is that p is both defined and true. In certain situations, results of this kind are desirable: for instance, when proving total correctness of programs. (Notice that, in Weak Logic, a result of this kind requires two independent proofs.)

Let the notation $A \sqsubseteq_{\text{STRICT}} B$ (read: A is a strict implementation of B) mean that $A \sqsubseteq B$, and, in addition, that A satisfies all the definedness properties of B . This corresponds to the traditional abstraction concept. If $T' \sqsubseteq_{\text{STRICT}} T$, the following rules hold.

$$\frac{T' \sqsubseteq_{BL} q}{T' \sqsubseteq_{BL} q} \quad \frac{T' \sqsubseteq_{WL} q}{T' \sqsubseteq_{WL} q}$$

where q' is q with each T -function replaced by the corresponding T' -function, and where the subscript BL (WL) indicates extended Base Logic (Weak Logic). If $T' \sqsubseteq T$, the following rule holds (in addition to the one given in the first part of the introduction):

$$\frac{T' \sqsubseteq_{WL} q}{T' \sqsubseteq_{BL} q}$$

2. Weak Logic

In this section, we will present Weak Logic, a first order logic that allows partial functions and partial predicates. As explained above, Weak Logic is designed to serve as a suitable basis for program reasoning. The ideas behind Weak Logic are general and therefore independent of a particular programming language (provided it has no call by name or lazy evaluation). However, in order to give a precise and

simple presentation, we shall use our own sample expression language, which resembles that of the specification and programming language ABEL [Dah 84]. Each function (and predicate) is introduced by a signature, identifying its domain and range. User-defined functions are characterized by user-defined axioms (see section 2.1). The expression language has a small but expressive kernel. By a set of rewrite rules, we may define abbreviations and more elaborate notation (see section 2.2). Since the set of rewrite rules defines the semantics of non-kernel formulas, it suffices to define validity and provability for kernel formulas: The definedness-operator is defined recursively over the kernel expression language (see section 2.3).

The validity concept of Weak Logic is defined by means of an interpretation of kernel formulas into formulas of Base Logic, which is a version of traditional first order logic that allows strongly defined formulas only (see sections 2.4 and 2.5). Finally, the provability concept of Weak Logic is given by a set of axioms and proof rules. (See section 2.6.)

Notational conventions

Throughout this paper, we shall use the following notation:

e, t denote expressions (or lists of expressions)

p, q denote formulas (i.e., boolean expressions)

f, g denote function symbols (including constants)

T, B denote type symbols (or type products)

x, y denote variables (or lists of variables)

$\vdash$ and $\models$ denote provability and validity, respectively.

All variables and function symbols will start with a lower case letter, whereas type symbols will start with an upper case letter. Meta functions are spelled by upper case letters, for instance *DEF*. Any occurrence of the meta expression *DEF* inside an expression denotes the formula that the *DEF*-operator associates to e . The meta expression q^x denotes q with all free occurrences of x replaced by e , renaming bound variables in q if they coincide with free variables of e .

The notation $p \vdash q$, where p may be a (possibly empty) list of formulas, means that q is provable under the assumption(s) p ; i.e., there is a sequence of formulas where each formula is either an

assumption, an axiom, or follows from previous formulas by a proof rule, and where the last formula is q . Also q may be a list of formulas:

$p \vdash (q_1, \dots, q_n)$ means $p \vdash q_1$, and $\dots$, and $p \vdash q_n$.

Similarly, the notation $p \models q$ is extended such that both p and q may be lists of formulas.

By definition, provability and validity are reflexive and transitive relations, and satisfy the following basic principles (shown below for provability):

$p \vdash q$, provided the list p contains each formula in q

if both $q \vdash p$ and $q' \vdash p$

A proof rule is written on the form

$$\frac{\dots; r \vdash p, \dots}{q}$$

which means that from $\dots$ and $(s; r) \vdash p$, and $\dots$ we may infer $s \vdash q$, where s is a formula (list) that satisfies the restrictions of the proof rule (if any). (The all-introduction rule below has a restriction on assumptions.) For example, the rule $\frac{p}{p} \vdash q$ implies $p \vdash q$ (since $p \vdash p$ holds).

Interpretation of Assumptions

It is possible to interpret $p \models q$ as either:

- i) $\forall x: T [p] \models \forall x: T [q]$, or
- ii) $\forall x: T [p] \models \forall x: T [q]$

where x are the free variables of p , and T the type of x . It is assumed here that p and q are (strongly) defined formulas. For our purposes the latter semantics is more practical. For instance, consider the statement:

$$x = y \models \text{succ}(x) = \text{succ}(y)$$

With semantics (i), it would be trivially true (because the assumption is false). With semantics (ii), it would express a reasonable fact. We shall therefore adopt semantics (ii) for assumptions. This means that axioms should be formulated as closed assumptions, whereas assumptions in a proof can simply be formulated as open assumptions.

Let p and q above be partly defined formulas. The semantics of $p \models q$ is now more complex. As will be explained later, the intuitive semantics in Base Logic, extended Base Logic, and Weak Logic, respectively, can be expressed by:

- $\forall x: T [(p \text{ is defined}) \text{ and } ((p \text{ is true}) \text{ implies } (q \text{ is defined and true}))]$ (Base-Logic)
- $\forall x: T [(p \text{ is defined and true}) \text{ implies } (q \text{ is defined and true})]$ (Extended-Base-Logic)
- $\forall x: T [(p \text{ is undefined or true}) \text{ implies } (q \text{ is undefined or true})]$ (Weak-Logic)

assuming the informal is-predicate is non-strict, and never undefined.

2.1. Introduction of User-defined Functions and Types

In order to reason about user-defined functions and types, we need a formal system where a user can introduce function and type symbols and axioms, in addition to the pre-defined boolean operators, axioms, and proof rules. We shall first restrict ourselves to "unconstrained" types. The last part of this section will show how (a simple notion of) constrained types can be allowed.

Each pre-defined and user-defined function must be introduced by a signature, which identifies its name, domain, range, and "kind", which is either total or partial. For each partial function, f , there is an associated predicate, denoted $\text{in } f$. These domain predicates enable us to express and reason about definedness properties of expressions. The semantics of user-defined functions, and their associated domain predicates, is characterized through user-defined axioms. An axiom is a closed formula that may be used as an assumption. If T is a system of user-defined types, functions, and axioms, and if q is a formula referring to functions in T or pre-defined functions only, then $T \vdash q$ ($T = q$) denotes that q is provable (valid) with the axioms of T as assumptions. (For more details, see for instance [Owe 80].)

A formal system over Weak Logic consists of a syntactic part, (i) and (ii) below, and a semantic part, (iii) and (iv) below:

- i) a set of type symbols called base types. The only pre-defined base type is *Boolean*.
- ii) a set of function symbols and for each function symbol f a signature of form

$$KIND \ f : D \rightarrow R$$

where *KIND* is either the word *total* or *partial*, and where D and R are products of base types, called the domain and range of f , respectively. If *KIND* is *total*, f is called a total function, otherwise, a partial function.

For each partial function f , there is an associated total function,

$$total \text{ in } f : D \rightarrow Boolean$$

called the domain predicate of f .

For each base type B , there is an associated total equality predicate:

$$total = : B * B \rightarrow Boolean$$

- iii) a set of user-defined axioms and definitions (see below).
- iv) a set of pre-defined axioms and proof rules (described in section 2.6).

An axiom is a closed formula. A definition has the form

$$def \ f(x) \equiv e$$

which is equivalent to the two axioms

$$\forall x : D \ [f(x) = e]$$

$$\forall x:D [x \text{ in } f = DEF[e]]$$

provided f is partial. If f is total, the latter axiom reduces to $\vdash x:D [DEF[e]]$, which states that e must be defined. (Otherwise, the definition yields an inconsistent system.) Note that the definition defines f as well as its associated domain predicate, if any. The symbol $\equiv$ (equal by definition) represents non-strict equality, i.e. equal in values and in definedness, whereas the symbol $=$ (executable equality) represents strict equality (see also section 3.3).

The base types represent distinct sets of values, each with at least two elements. For instance, the type boolean has two values, denoted *True* and *False*. *True* and *False* are pre-defined boolean constants:

$$total \ True : \neg Boolean$$

$$total \ False : \neg Boolean$$

In addition, boolean has the following pre-defined operations:

$$total \ not : Boolean \rightarrow Boolean$$

$$total \ op : Boolean * Boolean \rightarrow Boolean$$

where op is one of $=$, and , or , $implies$. The user shall not specify the type boolean or signatures of the pre-defined boolean operators, equality, and domain predicates. Pre-defined axioms and proof rules for the pre-defined functions, as well as for if-constructs and quantifiers, are found in section 2.6 and Appendix A. In order to extend the set of pre-defined concepts, commonly used functions and types could be stored in a library (or data base).

Note:

The domain predicate of a partial constant (function) c is itself a constant. In the case where $()$ in c is *True*, the constant could have been introduced as a total constant (see also section 2.3). In the case where $()$ in c is *False*, the constant names an erroneous situation, such as $1 / 0$, *underflow*, or *overflow*. Since a constant either denotes a defined value or an undefined value, it would be appropriate to state this in the signature, for instance as

$$total \ c : \rightarrow T \quad or \quad error \ c : \rightarrow T, \text{ respectively (see also section 4.1).}$$

Constrained types

The remaining part of this section will show how it is possible to allow semantically constrained types (subtypes). We shall restrict ourselves to subtypes of base types. Subtypes of subtypes can be introduced in a similar fashion. First, we introduce a new set of type symbols, called constrained types, such that, for each constrained type T , there is:

- a) an associated base type, denoted $Base_T$
- b) an associated total function

total in $T:Base_T \rightarrow Boolean$

called the domain predicate of T .

For a base type B , we define $Base_B$ as B , and

$def\ x\ in\ B \equiv True$

which expresses that base types have no semantic constraint (it is possible to check syntactically whether e in B). Note that an expression of type T is also an expression of type $Base_T$; therefore, an equality

function on type T is not needed.

For each type T (constrained or unconstrained), there is a set of associated functions. A function is

associated with, at most, one type, and only if this type occurs in its domain or range. We write f_T to

indicate that f is associated to T .

Example:

Assume $Stack$ is a base type with associated functions $push, pop, length$ (see section 3.3). We may introduce $BoundedStack$ as a subtype of $Stack$ by defining the constraint

$def\ s\ in\ BoundedStack \equiv length(s) \leq Max$

and letting $BoundedStack$ inherit the functions and axioms of $Stack$, using the following transforma-

tion method: Each (total or partial) user-defined $Stack$ function, for instance

total $push: Stack*Item \rightarrow Stack$

can be mechanically transformed to a partial $BoundedStack$ function:

partial $push: BoundedStack*Item \rightarrow BoundedStack$

$def\ push(s, x) \equiv if\ push_{stack}(s, x)\ in\ BoundedStack\ then\ push_{stack}(s, x)$

else $error_{BoundedStack}$

where the meaning of signatures with constrained types is explained below, and where

$error_{BoundedStack}$ is a $BoundedStack$ constant which is undefined, say

partial $error: \rightarrow BoundedStack$

$def\ in\ error \equiv False$

This implies that the domain predicate associated with $push_{BoundedStack}$ is defined by:

total in $push: Stack*Item \rightarrow Boolean$

$def\ (s, x)\ in\ push \equiv s\ in\ BoundedStack\ and\ push_{stack}(s, x)\ in\ BoundedStack$

In the case where $push_{stack}$ is partial, one must add $(s, x) \text{ in } push_{stack}$ to the right hand side of the last definition.

Assume the BoundedStack s has length Max . We now have

$(s \text{ in } Stack)$ and $(s \text{ in } BoundedStack)$, as well as $(push_{stack}(s, x) \text{ in } Stack)$; but not $(push_{stack}(s, x) \text{ in } BoundedStack)$. Furthermore, $(push_{BoundedStack}(s, x))$ is undefined.

In this example, the $push$ -function of $BoundedStack$ is a partial implementation of the $push$ -

function of $Stack$: i.e., $push_{BoundedStack} \sqsubseteq push_{stack}$

Furthermore, $BoundedStack$ is a partial implementation of $Stack$. (See also section 3.3.)

□

We shall now show the meaning of signatures with constrained types. Consider the signature below:

$KIND \quad f : D \rightarrow R$ (f-signature)

where D and/or R may be constrained. This function is defined for arguments in D only, and whenever defined, the function value is inside R . If f is total, f is defined for all arguments in D . (As a consequence, f is strictly total only if D is not constrained.) More formally, this signature is defined by the following transformation rules:

When R is constrained, the f-signature above is equivalent to

$KIND \quad f : D \rightarrow Base_R$

with the additional axiom

$\forall x : D \mid f(x) \text{ in } R$

When D is constrained, the f-signature above is equivalent to

$partial \quad f : Base_D \rightarrow R$

and, in addition, the first axiom below if f is partial, and the second, if f is total (according to the original f-signature):

$\forall x : Base_D \mid x \text{ in } f \text{ implies } x \text{ in } D$ (f-partial)

$\forall x : Base_D \mid x \text{ in } f = x \text{ in } D$ (f-total)

This transformation assumes that a definition of f , if any, has already been transformed to axioms as described above, and as demonstrated by the example below.

Example:

If D and/or R are constrained, we may conclude that the definition

$def \quad f(x) \equiv e$

is equivalent to the following axioms, where the axiom marked f-partial (f-total) applies only when f is partial (total):

$$\begin{aligned} \text{(f-partial)} \quad & \forall x:Base_D [x \text{ in } f = (x \text{ in } D \text{ and } DEF[x])] \\ \text{(f-total)} \quad & \forall x:Base_D [x \text{ in } f = x \text{ in } D \text{ and } \forall x:D, DEF[x]] \end{aligned}$$

$$\forall x:D [f(x) \text{ in } R]$$

$$\forall x:D [f(x) = e]$$

In the case where f is partial, f is defined for the subset of D where e is defined. In the case where f is total, f is defined for all values in D , and it is required that e be defined and have a value in R , for every x in D .

□

2.2. Expression Language

Our expression language is defined by a kernel and a set of rewrite rules. The rewrite rules define the semantics of non-kernel expressions by means of kernel expressions. The four constructs below define the kernel expression language recursively:

- i) variable x
- ii) functional term $f(e)$, where e is an expression list, say $e_1, \dots, e_n$ ($n \geq 0$)
- iii) if - expression $\text{if } q \text{ then } e_1 \text{ else } e_2$
- iv) universal quantification $\forall x:T [q]$

For notational conveniences, we will write some functional terms with infix and dot notation, and use overloading of functions (and constants) when no confusion results. For instance, we shall write

$$e_1 = e_2, \quad e_1 + e_2, \quad p \text{ and } q, \quad \text{not } q, \quad (e) \text{ in } f, \quad e_1.push(e_2)$$

rather than

$$equal_B(e_1, e_2), \quad plus_B(e_1, e_2), \quad and_{Boolean}(p, q), \quad not_{Boolean}(q), \quad in_f(e), \quad push_B(e_1, e_2)$$

where B is the base type of e_1 . With respect to binding priorities, we use the following order:

, not, and, or, =, implies, where "dot" binds the strongest, and we will also use parentheses to specify the nesting of infix operators.

$p \text{ implies then } q \rightarrow \text{ if } p \text{ then } q \text{ else } True$

$p \text{ and then } q \rightarrow \text{ if } p \text{ then } q \text{ else } False$

The following rules are used to abbreviate boolean if-expressions:

Abbreviated If-forms

where e_1 is a (new) non-kernel construct (which must be syntactically distinct from old constructs), and where e_2 is a kernel expression, or an expression that can be rewritten as a kernel expression by means of previously defined rewrite rules.

$$e_1 \rightarrow e_2$$

rewrite rule [Knu 70] has the form

In this section, we introduce some examples of non-kernel constructs by means of rewrite rules. A

2.2.1. Non-kernel Expressions

are therefore not considered kernel expressions.

$plus_{integer}(0, EmptyStack), \text{ if } 0 \text{ then } \dots \text{ else } \dots$

formed expressions such as

We shall require that all expressions be syntactically well formed, i.e., syntactically meaningful. III-

$isEmpty(EmptyStack)$

$s.add(x).delete(y) : \text{ if } x = y \text{ then } s.delete(y) \text{ else } s.delete(y).add(x)$

$A : Stack \rightarrow x : Item : s.push(x).pop = s$

be on universal form. The following examples of kernel formulas have universal form:

A formula of form $\forall x_1:T_1 \mid \forall x_2:T_2 \mid \dots \forall x_n:T_n [q[\dots]]$, where q is quantifier free, is said to

occur.)

boolean constants *True* and *False*. (Since variables start with a lower-case letter, no confusion should

shall write constant symbols with a leading upper-case letter or a digit, as, for instance, the pre-defined

Note that a constant has the form $f()$. In order to avoid writing empty sets of parentheses, we

must be "declared" by appropriate assumptions.

the first assumption states that the type of s is *Set*. As a consequence, all free variables of a theorem

$(s \text{ in } Set : x \text{ in } Item) \rightarrow s.add(x).add(x) = s.add(x)$

(free) variables by open assumptions. For instance, in the theorem

required that the type of each variable is declared. For our purpose, it suffices to express the type of

In order to allow overloading rules that resemble those of a modern programming language, it is

$p \text{ or else } q \rightarrow \text{if } p \text{ then True else } q$

Note that these operators are all evaluated from left to right: i.e., when p is undefined, the abbreviated if-forms are also undefined. In other words, they are strict with respect to the first operand, and non-strict with respect to the second operand. (The operators *and then* and *or else* are often called *cand* and *cor*, respectively.)

Non-strict Boolean Operators

It is possible to introduce non-strict versions of "and", "or", and "implies", denoted

' , $\vee$, and $=>$, respectively, which are non-strict with respect to both operands. These operators will be convenient within formal reasoning.

$p \wedge q \rightarrow \text{if DEF}[p] \text{ then } (p \text{ and then } q) \text{ else } (q \text{ and then } p)$

$p \vee q \rightarrow \text{if DEF}[p] \text{ then } (p \text{ or else } q) \text{ else } (q \text{ or else } p)$

$p => q \rightarrow (not\ p) \vee q$

Notice that these definitions are not executable. (However, an executable implementation can be realized through concurrency.) When one or both arguments are undefined, the non-strict boolean operators are undefined, except in the following cases:

$\wedge$ is defined and false when one argument (either one) is defined and false;

$\vee$ is defined and true when one argument is defined and true, and

$=>$ is defined and true when the first argument is defined and false,

or when the second argument is defined and true.

Note that $\wedge$ and $\vee$ are commutative.

Abbreviated Quantified Formulas

Existential quantifiers are defined by the rewrite rule:

$\exists x:T[q] \rightarrow not\ \forall x:T[not\ q]$

A list of formulas can be quantified according to the rule

$\forall x:T[q_1, \dots, q_n] \rightarrow \forall x:T[q_1] \text{ and } \dots \text{ and } \forall x:T[q_n]$

As will become clear, the quantification above is stronger than $\forall x:T[q_1 \text{ and } \dots \text{ and } q_n]$.

Nesting of quantifiers can be abbreviated according to the rules:

$\forall x_1, \dots, x_n:T[q] \rightarrow \forall x_1:T[\dots] \vee \dots \vee \forall x_n:T[q]$

$\forall x_1:T_1, \dots, x_n:T_n[q] \rightarrow \forall x_1:T_1[\dots] \vee \dots \vee \forall x_n:T_n[q]$

If each x_i is a single variable, the latter formula may also be written as $\forall (x_1, \dots, x_n) : T_1 * \dots * T_n \vdash q$. Similar rules apply to existential quantifiers as well.

Expression Lists

Equality over expression lists is defined by the rule

$$(e_1, e_2, \dots, e_n) = (e'_1, e'_2, \dots, e'_n) \rightarrow e_1 = e'_1 \text{ and } e_2 = e'_2 \text{ and } \dots \text{ and } e_n = e'_n$$

2.3. The definedness-operator

This section defines the definedness-operator for kernel expressions by means of structural induction.

For an expression e , $DEF[e]$ (sometimes written $DEF_{WL}[e]$) is a formula over the free variables of e . The intuitive meaning of $DEF[e]$ is: Is it true or false that e is defined?

$$DEF \vdash True$$

$$DEF \vdash f(e) \equiv \begin{cases} DEF \vdash e & \text{if } f \text{ is total} \\ DEF \vdash e \text{ and then } (e) \text{ in } f & \text{if } f \text{ is partial} \end{cases}$$

$$\text{where } DEF[e_1, \dots, e_n] \equiv DEF \vdash e_1 \text{ and } \dots \text{ and } DEF \vdash e_n$$

$$DEF \vdash \text{if } p \text{ then } e_1 \text{ else } e_2 \equiv DEF \vdash p \text{ and then if } p \text{ then } DEF \vdash e_1 \text{ else } DEF \vdash e_2$$

$$DEF \vdash \forall x : T [q] \equiv True$$

An expression e is said to be defined if $\models DEF[e]$, and is said to be undefined if $\models \text{not } DEF[e]$. In section 3, it will be proved that $DEF[e]$ is defined. Note that $DEF[q_1^*]$ implies $DEF[q]$, and note that they are the same if e is defined. Also note that the above definition allows the arguments of a functional term to be evaluated in parallel.

Examples:

Let op be one of $=$, and , or , $implies$.

$$DEF \vdash e_1 \text{ op } e_2 \equiv DEF \vdash e_1 \text{ and } DEF \vdash e_2$$

For non-kernel constructs, we may derive the following definedness properties:

$$DEF \vdash x : T [q] \equiv True$$

$$DEF \vdash p \text{ and then } q \equiv DEF \vdash p \text{ and then } (p \text{ implies then } DEF \vdash q)$$

$$DEF \vdash p \text{ implies then } q \equiv DEF \vdash p \text{ and then } (p \text{ implies then } DEF \vdash q)$$

$$DEF \vdash p \text{ or else } q \equiv DEF \vdash p \text{ and then } (not \ p \text{ implies then } DEF \vdash q)$$

$$DEF \vdash p \vee q \equiv (DEF \vdash p \text{ and then } p) \text{ or } (DEF \vdash q \text{ and then } q) \text{ or } (DEF \vdash p \text{ and } DEF \vdash q)$$

$$DEF[p > q] = (DEF[p \text{ and then not } p] \text{ or } (DEF[q \text{ and then not } q] \text{ or } (DEF[p \text{ and } DEF[q]] \\ DEF[p \wedge q] = (DEF[p \text{ and then not } p] \text{ or } (DEF[q \text{ and then not } q] \text{ or } (DEF[p \text{ and } DEF[q]]))$$

(The three last equations are slightly simplified.)

Given the following specifications:

$$total \ length : Stack \rightarrow Nat \ 0$$

$$partial \ pop : Stack \rightarrow Stack$$

$$def \ s \ in \ pop \equiv not \ s.length = 0$$

we may conclude

$$DEF \ not \ s.length = 0 \ implies \ then \ s.pop.length < s.length =$$

$$not \ s.length = 0 \ implies \ then \ not \ s.length = ()$$

where the right hand side ca5518 DEF-operator as a meta function. It would, how-

ever, be possible to introduce the DEF-operator as a construct of the Weak Logic expression language, taking the above definition as a set of axioms. The advantage would be that the DEF-operator would be available in the expression language itself. The disadvantage would be that the proof rule fulfilling Requirement 2 of the introduction must be restricted to formulas without occurrences of the DEF-operator, and the meta concept of substitution would be more complicated.

2.4. Base Logic

As Base Logic, we choose a version of first order logic with a trivial semantics of definedness (see also section 1.3). For simplicity, we shall use the expression language introduced above. In Base Logic, a partial function may only be applied to arguments for which it is defined. This is enforced by a semantic restriction on the set of acceptable (meaningful) formulas: Only strongly defined formulas are accepted in Base Logic (see below). In contrast to Weak Logic, quantifiers are strict in Base Logic. The only non-strict kernel construct is the if-construct. The non-strict, non-kernel constructs include the abbreviated if-forms and the non-strict boolean operators.

Examples:

The following formulas are acceptable in Base Logic:

$$\forall x : Integer, not \ x = 0 \ implies \ then \ x / x = 1$$

$$not \ s.length = 0 \ implies \ then \ s.pop.length + 1 = s.length$$

$DEF[p] \text{ implies then } p$

$DEF[p] \text{ and then } p$

$DEF[p] = > p$

$DEF[p] \wedge p$

provided p is a formula without quantifiers. The three first formulas would not be acceptable if *implies then* was replaced by *implies*.

□

An expression e is said to be strongly defined if the formula (denoted) $DEF_{BL}[e]$ is valid in Base Logic, where the DEF_{BL} -operator is defined for kernel expressions exactly as the DEF_{WL} -operator above, except for quantifiers, which follow the rule:

$$DEF_{BL} \vdash x : T \mid q \equiv \forall x : T \mid DEF_{BL}[q]$$

It can be proved that the formula (denoted) $DEF_{BL}[e]$ is itself strongly defined, which means that one can reason within Base Logic itself about strongly definedness of expressions. Consequently, validity and provability is defined for a formula p , and a list of assumptions q , if and only if

i) the assumptions are strongly defined; i.e.,

$$\models_{BL} DEF_{BL}[q]$$

ii) p is strongly defined assuming the assumptions are valid; i.e.,

$$q \models_{BL} DEF_{BL}[p]$$

For instance, (n in Integer, not $n = 0$) $\models_{BL} n / n = 1$ is acceptable in Base Logic. Note that the formula (denoted) $DEF_{WL}[e]$ may have occurrences of quantifications that are not acceptable in Base Logic. Also note that the basic principle $p \vdash_{BL} p$ requires that p be strongly defined, since Base Logic is restricted to strongly defined assumptions.

Below is a consistent and complete (in the sense that provability coincides with validity) set of

axioms and proof rules for Base Logic. A proof of consistency and completeness is beyond the scope of

this paper since similar proofs can be found elsewhere, see for instance [Ren 83, Pra 65, Sch 67].

Axiom schemas for Base Logic

$$\forall x : T \mid DEF_{BL}[DEF_{BL}[e]] \quad \text{(definedness)}$$

where x is the list of the free variables in e , and where T is the product of the corresponding base types.

Proof Rules of Base Logic

In addition to the tautology rule, there are introduction and elimination rules for boolean if-

expressions and universal quantifications:

$$\frac{}{\vdash DEF_{BL}[q]} \quad \text{(tautology)}$$

provided q is a tautology of propositional logic with if-expressions and equality (satisfying reflexivity, symmetry, transitivity, and substitutivity) over all types.

$$\begin{aligned} \text{(if-I)} \quad & \frac{p \vdash q_1, \text{ not } p \vdash q_2}{p \vdash \text{if } p \text{ then } q_1 \text{ else } q_2} \\ \text{(if-E1)} \quad & \frac{\vdash q_1}{\vdash p, \text{ if } p \text{ then } q_1 \text{ else } q_2} \\ \text{(if-E2)} \quad & \frac{\vdash q_2}{\vdash \text{not } p, \text{ if } p \text{ then } q_1 \text{ else } q_2} \end{aligned}$$

where the premise may not have any assumption (except x in T) about x .

$$\text{(all-E)} \quad \frac{\vdash \forall x:T, p, \vdash e \text{ in } T}{\vdash p'_x}$$

Note that in each rule the conclusion is strongly defined, provided the premises are strongly defined

and are true.

Derived proof rules

As examples of derived proof rules, we show the following rules for strict and non-strict implication:

$$\text{(=>-E)} \quad \frac{\vdash p, \vdash p = > q}{\vdash q}$$

The above rule can also be derived for *implies then* and *implies*.

$$\text{(=>-I)} \quad \frac{\vdash p \vdash q}{\vdash p = > q}$$

The above rule also holds for *implies then*, but not for *implies*, which requires an additional premise in

the introduction rule:

$$\text{(implies-I)} \quad \frac{\vdash p \vdash q, \text{ not } p \vdash DEF_{BL}[q]}{\vdash p \text{ implies } q}$$

For the strict and non-strict and-operators, we may derive the following rules:

$$\text{(and-I)} \quad \frac{\vdash p \text{ and } q}{\vdash (p, q)}$$

The above rule also holds for *and then* and *or then*.

$$\frac{\vdash p \text{ and } q}{\vdash (p \cdot q)}$$

(and-E)

The above rule also holds for *and then* and *or then*.

Again, each rule satisfies that the conclusion is strongly defined, provided the premises are strongly defined. We may actually prove that Base Logic, with the proof rules above, satisfies the following

(meta) theorem:

$$\vdash p \text{ implies } DEF_{BL} p$$

That is, all provable formulas are (provably) strongly defined. (The proof is straightforward, using induction on the length of the proof of q . It suffices to prove that each proof rule satisfies the theorem.) Therefore, rather than restricting Base Logic to strongly defined formulas, we may easily extend Base Logic to allow undefined and partly defined formulas, with the effect that only strongly defined formulas will be valid and provable

2.4.1. Extended Base Logic

In the extended version of Base Logic, there is no restriction on the set of acceptable formulas, and a formula p will be provable (valid) if, and only if, p is both strongly defined and is provable in the restricted version of Base Logic. For instance, the so-called "Law of Excluded Miracle":

$$\vdash p \text{ or not } p$$

is no longer satisfied, since p is no longer restricted to strongly defined formulas.

Consider assumptions, the basic principle $p \vdash p$ must now be valid, even when p is undefined.

The only reasonable solution is to let $q \vdash p$ mean:

" q is strongly defined and true" $\vdash$ " p is strongly defined and true".

In the case where q is undefined, we have: $False \vdash p$, which is a valid result, and which should be provable. As a consequence of this modified semantics of assumptions, those rules above which use assumptions must be modified accordingly. The only basic proof rule of this kind is the if-introduction rule. It needs an additional premise stating that the if-test must be strongly defined:

$$\frac{\vdash DEF_{BL} p, p \vdash q 1, \text{ not } p \vdash q 2}{\vdash \text{ if } p \text{ then } q 1 \text{ else } q 2} \quad \text{(if-I)}$$

All the other basic proof rules of Base Logic are used in extended Base Logic without modification. The derived rules can be modified similarly. Extended Base Logic is consistent and complete (with the same axioms and basic proof rules as Base Logic, except for if-introduction). This result is quite obvious from the meta theorem given above.

2.5. Validity

This section defines the validity concept of Weak Logic by means of an interpretation of Weak Logic formulas into Base Logic formulas, where Base Logic is described above. A formula p is said to be valid in Weak Logic, denoted $\models_{WL} p$, if its interpretation is valid in Base Logic; i.e.,

$$\models_{WL} p \quad \text{if and only if} \quad \models_{BL} I[p]$$

where the interpretation operator I is defined below. Similarly, p is said to be valid under the assumption q if the interpretation of p is valid in Base Logic under the assumption of the interpretation of q ; i.e.,

$$q \models_{WL} p \quad \text{if and only if} \quad I[q] \models_{BL} I[p]$$

We extend the definition above to lists of formulas (both p and q may be lists) by defining

$$I([p_1, \dots, p_n]) \equiv I[p_1], \dots, I[p_n]$$

The I -operator takes any (well formed) formula into a strongly defined formula. We define I for

kernel formulas by

$$I[q] \equiv IQ[Wq]$$

where the meta functions W and IQ interpret partly defined formulas and (universal) quantifiers, respectively, in a weak fashion, as defined below. The W -operator takes a formula into a defined formula:

$$W[p] \equiv (DEF[p] \text{ implies then } p) \quad \text{(W-definition)}$$

The IQ -operator takes a defined kernel formula into a strongly defined kernel formula, by taking a weak interpretation of all (sub)formulas immediately following a universal quantifier. The IQ -operator is defined by structural induction over kernel expressions:

$$IQ[x] \equiv x$$

$$IQ[f(e)] \equiv f(IQ[e]) \quad \text{where} \quad IQ[e_1, \dots, e_n] \equiv IQ[e_1], \dots, IQ[e_n]$$

$$IQ[\text{if } p \text{ then } e_1 \text{ else } e_2] \equiv \text{if } IQ[p] \text{ then } IQ[e_1] \text{ else } IQ[e_2]$$

$$IQ[\forall x:T|q] \equiv \forall x:T|I[q]$$

(IQ-definition)

In order to prove that for a given kernel formula p , $I[p]$ is a meaningful (i.e., strongly defined) formula of Base Logic, it suffices to prove that $W[p]$ is a defined kernel formula, and secondly, that

$IQ[Wp]$ is a strongly defined kernel formula. Both these facts are obvious from the definitions of W

and IQ above.

Furthermore, $IQ[DEF[e]] \equiv DEF[IQ[e]]$, can easily be proved by structural induction on e . As

a result, W and IQ can be applied in any order:

$$I[q] \equiv IQ[Wq] \equiv IQ[DEF[q]] \text{ implies then } q \equiv (DEF[IQ[q]] \text{ implies then } IQ[q]) \equiv W[IQ[q]]$$

2.5. Validity

This section defines the validity concept of Weak Logic by means of an interpretation of Weak Logic formulas into Base Logic formulas, where Base Logic is described above. A formula p is said to be valid in Weak Logic, denoted $\models_{WL} p$, if its interpretation is valid in Base Logic; i.e.,

$$\models_{WL} p \quad \text{if and only if} \quad \models_{BL} I[p]$$

where the interpretation operator I is defined below. Similarly, p is said to be valid under the assumption q if the interpretation of p is valid in Base Logic under the assumption of the interpretation of q ; i.e.,

$$q \models_{WL} p \quad \text{if and only if} \quad I[q] \models_{BL} I[p]$$

We extend the definition above to lists of formulas (both p and q may be lists) by defining

$$I[p_1, \dots, p_n] \equiv I[p_1], \dots, I[p_n]$$

The I -operator takes any (well formed) formula into a strongly defined formula. We define I for kernel formulas by

$$I[q] \equiv IQ[W;q]$$

(I-definition)

where the meta functions W and IQ interpret partly defined formulas and (universal) quantifiers, respectively, in a weak fashion, as defined below. The W -operator takes a formula into a defined formula:

$$W[p] \equiv (DEF \mid p \text{ implies then } p) \quad \text{(W-definition)}$$

The IQ -operator takes a defined kernel formula into a strongly defined kernel formula, by taking a weak interpretation of all (sub)formulas immediately following a universal quantifier. The IQ -operator is

defined by structural induction over kernel expressions:

$$IQ[x] \equiv x$$

$$IQ[f(e)] \equiv f(IQ[e]) \quad \text{where} \quad IQ[e_1, \dots, e_n] \equiv IQ[e_1], \dots, IQ[e_n]$$

$$IQ[\text{if } p \text{ then } e_1 \text{ else } e_2] \equiv \text{if } IQ[p] \text{ then } IQ[e_1] \text{ else } IQ[e_2]$$

$$IQ[\mid \forall x:T \mid q \mid] \equiv \mid \forall x:T \mid I[q] \mid \quad \text{(IQ-definition)}$$

In order to prove that for a given kernel formula p , $I[p]$ is a meaningful (i.e., strongly defined) formula of Base Logic, it suffices to prove that $W[p]$ is a defined kernel formula, and secondly, that

$IQ[W[p]]$ is a strongly defined kernel formula. Both these facts are obvious from the definitions of W

and IQ above.

Furthermore, $IQ[DEF[e]] \equiv DEF[IQ[e]]$, can easily be proved by structural induction on e . As a result, W and IQ can be applied in any order:

$$I[q] \equiv IQ[W[q]] \equiv IQ[DEF[q]] \equiv IQ[DEF[IQ[q] \text{ implies then } q]] \equiv (DEF \mid IQ[q] \text{ implies then } IQ[q]) \equiv W[IQ[q]]$$

Note:

A consequence of our definition of I is that universal quantifiers can be said to have a "weak" semantics whereas existential quantifiers can be said to have a "strong" semantics.

$$I \vee x : T [q] = \vee x : T [DEF IQ [q] \text{ implies then } IQ q]$$

$$I \equiv x : T [q] =$$

$$\begin{aligned} I \text{ not } \vee x : T [\text{not } q] &= \text{not } \vee x : T [DEF IQ [\text{not } q] \text{ implies then } IQ [\text{not } q]] = \\ \text{not } \vee x : T [DEF IQ [q] \text{ implies then not } IQ [q]] &= \\ \text{not } \vee x : T [\text{not } (DEF IQ [q] \text{ and then } IQ [q])] &= \\ \equiv x : T [DEF IQ [q] \text{ and then } IQ [q]] \end{aligned}$$

Proof Rules of Weak Logic

This section defines the provability concept of Weak Logic by means of a set of proof rules. In addition to the Tautology rule, there are introduction and elimination rules for implication and for quantification. Because of the Tautology rule, no axioms are needed. However, if we do include a set of axioms that state enough basic facts about the predefined functions, then the rule of Tautology can be omitted. Such a set of axioms is presented in Appendix A.

2.6. Provability

q is a Tautology of propositional logic extended with if-expressions and equality

(Tautology)

$$\frac{p \vdash q}{p \text{ implies } q}$$

(implies-I)

$$\frac{DEF [q] \vdash (p, p \text{ implies } q, DEF [p])}{\vdash q}$$

(implies-E)

$$\frac{x \text{ in } T \vdash p}{\vdash \vee x : T [p]}$$

(all-I)

where the premise must not have any assumption (except $x \text{ in } T$) about x .

$$\frac{\vdash \vee x : T [p], \vdash e \text{ in } T}{\vdash p^2}$$

(all-E)

where either p must have universal form (see section 2.3), or $\vdash DEF [e]$ must be added as an additional premise.

Note that the rule of Tautology above is simpler than that of Base Logic -- no definedness premise is required since undefined formulas are trivially true in Weak Logic. The if-introduction rule of Base Logic can easily be derived. The if-elimination rule of Base Logic, however, is not valid in Weak Logic because the above Modus Ponens (implies-E) is different from that of Base Logic. The premises of the above Modus Ponens are, on one hand, harder to prove because there is an additional premise, but, on the other hand, easier to prove because there is an additional assumption for the premises. In contrast to Base Logic, all-E does not require that e be defined, provided p has universal form (or can be written on universal form). The following example shows that the restriction on the all-E rule is necessary: Let p

denote $\exists y: Nat \mid y = x$. Then $\forall x: Nat \mid p$ is valid. Now, $p_{1/0}$ which denotes $\exists y: Nat \mid y = 1/0$, is invalid.

Note:

The presented set of proof rules is based on the strict versions of the boolean operators. It is possible to give a consistent and complete (in the sense that provability coincides with validity) set of axioms and proof rules based on the non-strict versions of the boolean operators ($=$, $>$, $\wedge$, $\vee$), for instance by replacing the two implies rules by the following four rules:

$$\frac{\vdash (p, q)}{\vdash p \wedge q} \quad \frac{\vdash p \wedge q}{\vdash p} \quad \frac{\vdash (q, p)}{\vdash p \vee q} \quad \frac{\vdash p \vee q}{\vdash q}$$

$$\frac{\vdash (q, not\ q)}{\vdash not\ q} \quad \frac{\vdash not\ DEF[q]}{\vdash DEF[q], not\ q} \quad \frac{\vdash q}{(DEF[q], not\ q) \vdash False}$$

This would make it natural to introduce the non-strict boolean operators as kernel functions. With a set of logical axioms similar to those in Appendix A, the Tautology rule could be omitted (as before).

Derived proof rules

A few derived proof rules are showed in this section (their proofs are left to the reader):

$$\frac{\vdash p \text{ implies } q\ 1, \vdash not\ p \text{ implies } q\ 2}{\vdash \text{ if } p \text{ then } q\ 1 \text{ else } q\ 2} \quad \text{(if-I)}$$

$$\frac{\vdash \text{ if } p \text{ then } q\ 1 \text{ else } q\ 2}{\vdash (p \text{ implies } q\ 1, not\ p \text{ implies } q\ 2)} \quad \text{(if-E)}$$

$$\frac{\vdash q}{\vdash p \text{ or } q} \quad \text{(or-I)}$$

$$\frac{\vdash p\ 1 \text{ or } p\ 2, \vdash p\ 1 \vdash q, \vdash p\ 2 \vdash q}{\vdash q} \quad \text{(or-E)}$$

$$\frac{-}{-} \frac{(p, q)}{p \text{ and } q} \quad \text{(and-I)}$$

$$\frac{DEF[q] - (p \text{ and } q, DEF[p])}{-} \quad \text{(and-E)}$$

$$\frac{p \vdash \text{False}}{p \vdash \text{not } p} \quad \text{(not-I)}$$

$$\frac{p - q, \text{not } p - q}{-} \quad \text{(not-E)}$$

$$\frac{x \text{ in } T, (e \text{ in } T, q_1, DEF[q], DEF[e])}{-} \quad \text{(ex-I)}$$

$$\frac{- \equiv x: T, q, (q, x \text{ in } T, DEF[q]) \vdash p}{-} \quad \text{(ex-E)}$$

where x may not occur in p or any assumption (if any) of the second premise.

$$\frac{\vdash (q, \text{not } q)}{\vdash \text{not } DEF[q]} \quad \text{(DEF-I)}$$

$$\frac{\vdash q}{DEF[q] \vdash q} \quad \text{(DEF-E)}$$

$$\frac{\vdash q_1 = q_2}{\vdash q_1 = q_2} \quad \text{(equality)}$$

where either q must have universal form, or $DEF[e_1 - DEF[e_2]]$.

$$- p - q \quad \text{if and only if } \vdash (p \text{ implies } q, q \text{ implies } p) \quad \text{(Boolean-equality)}$$

We may derive the following proof rules for the non-strict boolean operators:

$$- p : > q \quad \text{if and only if } \vdash p \text{ implies then } q \text{ if and only if } \vdash p \text{ implies } q$$

$$- p \vee q \quad \text{if and only if } \vdash p \text{ or else } q \text{ if and only if } \vdash p \text{ or } q$$

$$- p \wedge q \quad \text{if and only if } \vdash (p, q)$$

$$\vdash p \text{ and then } q \text{ if and only if } \vdash (p, p \text{ implies } q)$$

In addition to the rules above, Weak Logic satisfies the "Law of Excluded Miracle":

$$\vdash q \vee \text{not } q \quad \text{(Excluded-Miracle)}$$

3. Main Properties of Weak Logic

In this section, we will show that Weak Logic is consistent and complete and satisfies the three

requirements given in the introduction.

3.1. Consistency and Completeness

In this section we will prove that Weak Logic (with the set of axioms and proof rules presented in

section 2.5) is complete and consistent, in the sense that a formula is provable if, and only if, it is valid.

First, we will prove some helpful (meta) theorems about Weak Logic and (non-extended) Base Logic.

Unless otherwise specified, $\vdash$, $\models$, and DEF mean $\vdash_{WL}$, $\models_{WL}$, and DEF_{WL} , respectively.

As already mentioned, we may prove that the formula $DEF[e]$ is defined:

$$\neg_{WL} DEF DEF[e] \quad (\text{Lemma-W1})$$

Proof:

We use structural induction on e . The most difficult cases are partial functions and if-expressions.

case $f(e)$, where f is partial:

$$DEF[DEF[f(e)]]$$

$$\equiv DEF[DEF[e] \text{ and then } e \text{ in } f] \quad , \text{ by definition of } DEF$$

$$\equiv DEF[DEF[e]] \text{ and then } (DEF[e] \text{ implies then } DEF[e] \text{ in } f) \quad , \text{ by definition of } DEF$$

$$\equiv DEF[DEF[e]] \text{ and then } (DEF[e] \text{ implies then } DEF[e]) \quad , \text{ since (in } f) \text{ is total}$$

Now, $\vdash DEF[DEF[e]]$ follows from the induction hypothesis,

and $\vdash DEF[e] \text{ implies then } DEF[e]$ is a tautology.

Finally, $\vdash DEF[DEF[f(e)]]$ follows by and then-introduction.

case if:

$$DEF[DEF[\text{if } p \text{ then } q1 \text{ else } q2]]$$

$$\equiv DEF[DEF[p] \text{ and then if } p \text{ then } DEF[q1] \text{ else } DEF[q2]] \quad , \text{ by definition of } DEF$$

$$\equiv DEF[DEF[p]] \text{ and then } (DEF[p] \text{ implies then } DEF[\text{if } p \text{ then } DEF[q1] \text{ else } DEF[q2]])$$

$$\equiv DEF[DEF[p]] \text{ and then } (DEF[p] \text{ implies then } (DEF[p] \text{ and then } DEF[DEF[q2]]))$$

$$\text{if } p \text{ then } DEF[DEF[q1]] \text{ else } DEF[DEF[q2]]$$

Now, $\vdash DEF[DEF[p]]$, by the induction hypothesis

$DEF[p] \vdash \text{if } p \text{ then } DEF[DEF[q1]] \text{ else } DEF[DEF[q2]]$, by the induction hypothesis and if-I

Finally, by implies then-introduction and and then-introduction, the desired result follows. $\square$

We may also prove that the formula $DEF_{BL}[e]$ is strongly defined:

$$\vdash_{WL} DEF_{BL} DEF_{BL}[e] \quad (\text{Lemma-W2})$$

Proof:

This proof is quite similar to the proof of Lemma-W1, except for the following case:

$$\begin{aligned} & DEF_{BL} [DEF_{BL} [\forall x : T [q]]] \\ & \equiv DEF_{BL} [\forall x : T [DEF_{BL} [q]]] \\ & \equiv \forall x : T [DEF_{BL} [DEF_{BL} [q]]] \end{aligned}$$

which follows from the induction hypothesis by the all-introduction rule.

□

The next lemma expresses that, in Base Logic, the interpretation of the definedness of an expression e equals the definedness of the interpretation of quantifiers in e :

(Lemma-B1)

$$\vdash_{BL} I, DEF\ e \ =\ DEF\ IQ\ [e]$$

Proof:

$$I\ [DEF\ [e]] \ =\ W\ [IQ\ [DEF\ [e]]] \ =\ W\ [DEF\ IQ\ [e]] \ =\ DEF\ [DEF\ IQ\ [e]] \implies \text{then } DEF\ [IQ\ [e]]$$

It suffices to prove: $\vdash_{BL} DEF\ [DEF\ IQ\ [e]]$

This proof can be done exactly as the proof of Lemma-W1, since the Weak Logic rules used there hold in Base Logic as well.

□

A formula that is provable in Base Logic is also provable in Weak Logic: i.e.,

(Theorem-1)

$$\vdash_{WL}\ q \quad \text{if} \quad \vdash_{BL}\ q$$

Proof:

We already know that

$$\vdash_{BL}\ q \implies \vdash_{BL}\ DEF_{BL}\ [q]$$

This means that, in each proof rule of Base Logic, we may assume that each premise is strongly defined (including assumptions of the premises also, if any), and we may conclude that the conclusion is also

strongly defined. Consequently, we may modify all the proof rules of Base Logic by adding premises stating that each premise is strongly defined, and adding another conclusion stating that the (original) conclusion is strongly defined. The provability of this modified Base Logic is the same as the provability for Base Logic.

The set of proof rules in the modified Base Logic is given below:

$$\frac{\vdash\ q}{\vdash\ DEF_{BL}\ [q]} \quad \text{(tautology)}$$

provided q is a tautology of propositional logic extended with if-expressions and equality (satisfying reflexivity, symmetry, transitivity, and substitutivity).

$$(if-I) \frac{DEF_{BL}[p] \vdash p \vdash (DEF_{BL}[q1] \cdot q1) \cdot not\ p \vdash (DEF_{BL}[q2] \cdot q2) \vdash (DEF_{BL}[if\ p\ then\ q1\ else\ q2])}{DEF_{BL}[p] \vdash p \vdash (DEF_{BL}[q1] \cdot q1) \cdot not\ p \vdash (DEF_{BL}[q2] \cdot q2)}$$

$$(if-E1) \frac{DEF_{BL}[p] \vdash p \vdash (DEF_{BL}[if\ p\ then\ q1\ else\ q2] \cdot if\ p\ then\ q1\ else\ q2)}{DEF_{BL}[p] \vdash p \vdash (DEF_{BL}[if\ p\ then\ q1\ else\ q2] \cdot if\ p\ then\ q1\ else\ q2)}$$

$$(if-E2) \frac{DEF_{BL}[p] \vdash p \vdash (DEF_{BL}[if\ p\ then\ q1\ else\ q2] \cdot if\ p\ then\ q1\ else\ q2)}{DEF_{BL}[p] \vdash p \vdash (DEF_{BL}[if\ p\ then\ q1\ else\ q2] \cdot if\ p\ then\ q1\ else\ q2)}$$

$$(all-I) \frac{x\ in\ T \vdash (DEF_{BL}[p] \cdot p)}{DEF_{BL}[p] \vdash p \vdash (DEF_{BL}[p] \cdot p)}$$

$$(all-E) \frac{DEF_{BL}[p] \vdash p \vdash (DEF_{BL}[p] \cdot p)}{DEF_{BL}[p] \vdash p \vdash (DEF_{BL}[p] \cdot p)}$$

where the all-introduction rule requires that the premise have no assumption (except x in T) about x .

Assume $\vdash_{BL} q$.

This means q can be proved by the axiom of Base Logic and by the rules above. Each rule in this set can easily be derived in Weak Logic. Secondly, the only axiom of Base Logic is provable in Weak Logic by

Lemma-W2 and all-introduction. Consequently, q is provable in Weak Logic.

□

A formula q is provable in Weak Logic if, and only if, the formula $(DEF[q] \text{ implies then } q)$ is provable

in Weak Logic; i.e.,

$$(Lemma-W3) \quad \vdash_{WL} q \text{ if and only if } \vdash_{WL} W[q]$$

where W is the meta function defined in section 2.4.

Proof:

Case only if: Obvious

Case if:

$$1. \vdash W[q]; \text{ i.e., } \vdash DEF[q] \text{ implies then } q, \text{ by assumption}$$

$$2. \vdash DEF[q] \text{ implies } q, \text{ by implies then-elimination}$$

$$3. DEF[q] \vdash DEF[q], \text{ by definition of provability}$$

$$4. \vdash DEF[DEF[q]], \text{ by Lemma-W1}$$

$$5. \vdash q, \text{ by 2, 3, 4, and Modus Ponens}$$

□

If a formula has the form $\dots \forall x:t[q] \dots$, then q may be replaced by $W[q]$ without affecting the (Weak Logic) provability of the total formula:

$$(Lemma-W4) \quad \vdash_{WL} p \text{ if and only if } \vdash_{WL} IQ[p]$$

Proof:

It suffices to prove (*) and (**) below:

(*)

$$\vdash e = IQ[e]$$

(**)

$$\vdash DEF[e] \text{ if and only if } \vdash DEF[IQ[e]]$$

This can be done by structural induction on the expression e . The proof of (**) is trivial. In the

proof of (*), the only non-trivial case is universal quantification, where we must prove:

(***)

$$\vdash \forall x:T \ q = \forall x:T \ W[IQ[q]]$$

The induction hypothesis is that (*) holds for any expression with the same (or less) structure as q .

1. (x in T , $\forall x:T \ W[IQ[q]] \vdash W[IQ[q]]$), by all-elimination

2. (x in T , $\forall x:T \ W[IQ[q]] \vdash IQ[q]$), by 1 and Lemma-W3

3. $DEF[q] \vdash (DEF[IQ[q]] \cdot IQ[q] \text{ implies } q)$

by (**), the induction hypothesis, and Boolean-equality

4. (x in T , $\forall x:T \ W[IQ[q]] \vdash q$), by 2, 3, and Modus Ponens

5. $\forall x:T \ W[IQ[q]] \vdash \forall x:T \ q$, by 4 and all-introduction

6. $\vdash \forall x:T \ W[IQ[q]] \text{ implies } \forall x:T \ q$, by 5 and implies-introduction

Similarly, we can prove $\vdash \forall x:T \ q \text{ implies } \forall x:T \ W[IQ[q]]$

Finally, (***) follows by the Boolean-equality rule.

□

We can now combine the two lemmas above.

(Theorem-2)

$$\vdash_{WL} I[q] \text{ if and only if } \vdash_{WL} q$$

Proof:

$$\vdash_{WL} I[q]$$

if and only if $\vdash_{WL} W[IQ[q]]$, by definition of I

if and only if $\vdash_{WL} IQ[q]$, by Lemma-W3

if and only if $\vdash_{WL} q$, by Lemma-W4

□

We can now prove that Weak Logic is complete.

(Theorem-3)

$$\vdash_{WL} q \text{ if } \models_{WL} q$$

Proof:

1. $\models_{WL} q$, by assumption

2. $\models_{BL} I[q]$, by 1, and by definition of $\models_{WL}$

3. $\models_{BL} I[q]$, by 2, since Base Logic is complete

4. $\vdash_{WL} I[q]$, by 3 and by Theorem 1 (since $I[q]$ is strongly defined)

5. $\vdash_{WL} q$, by 4 and Theorem 2.

We may now prove that Weak Logic is consistent.

(Theorem-4)

Proof:

By induction on the length of the proof of q , it suffices to prove that:

i) all axioms are valid, and

ii) each proof rule satisfies the following:

if all the premises are valid, then the conclusion is valid.

Since there are no explicit axioms for Weak Logic, it remains to prove (ii). Since Base Logic is consistent and complete, it suffices to prove that each proof rule (of Weak Logic) satisfies that

if the interpretation of each premise is provable in Base Logic,

then the interpretation of the conclusion is provable in Base Logic.

Let e' denote $IQ[e]$, for any expression e .

case Tautology

We may assume:

1. q is a tautology of propositional logic

We must prove: $\vdash_{BL} I[q]$; i.e., $\vdash_{BL} DEF[q']$ implies then q'

Let us assume:

2. $\vdash_{BL} DEF[q']$

It remains to prove: $\vdash_{BL} q'$

3. q' is a tautology of propositional logic , by 1 and definition of IQ

4. $\vdash_{BL} q'$, by 2, 3, and the tautology rule (of Base Logic).

case Modus Ponens:

We may assume:

1. $I[DEF[q]] \vdash_{BL} I[DEF[p]]$, $I[p]$ implies q

We must prove: $\vdash_{BL} I[q]$; i.e., $\vdash_{BL} DEF[q']$ implies then q'

2. $DEF[q'] \vdash_{BL} (DEF[p'] , DEF[p'] \text{ implies then } p')$, by 1 and Lemma B1

case implies-introduction:

We may assume:

1. $I \mid p \mid \vdash_{BL} I \mid q \mid$

We must prove: $\vdash_{BL} I \mid p \mid \text{ implies } q \mid$

Let us assume:

2. $\vdash_{BL} DEF \mid p \mid$ and $DEF \mid q \mid$

It remains to prove: $\vdash_{BL} p \mid \text{ implies } q \mid$

3. $\vdash_{BL} (DEF \mid p \mid \text{ and } DEF \mid q \mid) \text{ implies then } (p \mid \text{ implies } q \mid)$, by 1

4. $\vdash_{BL} p \mid \text{ implies } q \mid$, by 2, 3, and implies-I (of Base Logic).

case all-introduction

We may assume:

1. $I \mid x \text{ in } T \mid \vdash_{BL} I \mid p \mid$

We must prove: $\vdash_{BL} I \mid \forall x : T \mid p \mid$

2. $x \text{ in } T \vdash_{BL} I \mid p \mid$, by 1 (since in is total)

3. $\vdash_{BL} \forall x : T \mid I \mid p \mid$, by 2 and all-introduction

4. $\vdash_{BL} I \mid \forall x : T \mid p \mid$, by 3 and by definition of I.

case all-elimination

We may assume:

1. $\vdash_{BL} (I \mid \forall x : T \mid p \mid) , I \mid e \text{ in } T \mid$

and that p has universal form or that $\vdash_{BL} DEF \mid I \mid e \mid$

We must prove: $\vdash_{BL} I \mid p_x^e \mid$; i.e., $\vdash_{BL} DEF \mid (p_x^e) \mid \text{ implies then } (p_x^e) \mid$

Let us first assume that either p is quantifier free, or that $\vdash_{BL} DEF \mid I \mid e \mid$; i.e., $\vdash_{BL} DEF \mid e \mid$.

In either case, it is obvious that

$\vdash_{BL} (q_x^e) \mid$ if and only if $\vdash_{BL} q_x^e \mid$,

It suffices to prove: $\vdash_{BL} DEF \mid p_x^e \mid \text{ implies then } p_x^e \mid$

2. $\vdash_{BL} \forall x : T \mid DEF \mid p \mid \text{ implies then } p \mid$, by 1

3. $\vdash_{BL} DEF \mid e \mid \text{ implies then } e \mid \text{ in } T$, by 1

4. $DEF[e', \neg_{BL} DEF[p']]$ implies then p' , by 2, 3, and all-E
5. $DEF[e', \neg_{BL} DEF[p']]$ implies then p' , by definition of DEF and by tautology

In the case where p is quantifier free, it remains to prove:

$not\ DEF[e', \neg_{BL} DEF[p']]$ implies then p' : i.e.,

$(not\ DEF[e', \neg_{BL} DEF[p']]) \neg_{BL} p'$

Since $(x\ in\ T, DEF[p]) \neg_{BL} p$, by 2 and all-elimination,

it suffices to prove the following lemma:

$(x\ in\ T, not\ DEF[e', DEF[p']]) \neg_{BL} p = p'$

This lemma is easily proved using induction on the structure of p .

Secondly, assume p has the form $\forall y:Ty[q]$,

where y is a variable tuple and where q is quantifier free.

2. $(x\ in\ T, y\ in\ Ty) \neg_{BL} I[q]$, by 1 and all-E twice

3. $y\ in\ Ty \neg_{BL} \forall x:T[I[q]]$, by 2 and all-I

4. $y\ in\ Ty \neg_{BL} I[\forall x:T[q]]$, by 3 and definition of I

5. $y\ in\ Ty \neg_{BL} I[q']$, by 4 and the proof above (since q is quantifier free)

6. $\neg_{BL} \forall y:Ty[I[q']]$, by 5 and all-I

7. $\neg_{BL} I[(\forall y:Ty[q])']$, by 6 and definition of I .

□

We can now easily prove that Weak Logic is consistent and complete, relative to a set of axioms and

assumptions:

$p \vdash q$ if and only if $p \models q$

(Theorem-5)

Proof:

We may obviously assume that q is a list of only one formula. Let us first assume that p is a list of

one formula.

$p \models q$

if and only if $I[p] \models_{BL} I[q]$, by definition of validity

if and only if $I[p] \vdash_{BL} I[q]$, by completeness and consistency of BL

if and only if $\neg_{BL} I[p] \vdash I[q]$, by implies-I and -E (of BL)

if and only if $\neg_{BL} I[p] \vdash I[p] \vdash I[q]$, using BL-rules

if and only if $\neg_{BL} I[p] \vdash I[p] \vdash I[q]$, by consistency and completeness of BL

if and only if $\neg_{WL} I[p] \vdash I[q]$, by definition of validity

if and only if $\neg_{WL} I[p] \vdash I[q]$, by theorem 3 and 4

if and only if $I[p] \vdash_{WL} q$, using implies-1 and -E
 if and only if $p \vdash q$
 since $p \vdash I[p]$ and $I[p] \vdash p$, by theorem 2
 and since by definition $\vdash$ is transitive.

In the case where p has the form $(p_1, p_1, \dots)$, we may replace $I[p]$ in the proof above by

$(I[p_1] \text{ and } I[p_2] \text{ and } \dots)$.

□

3.2. Relationship to Base Logic

In this section, we will prove that Weak Logic satisfies Requirement 1; i.e., that for strongly defined formulas, provability in Weak Logic is the same as in Base Logic. Requirement 1 can be formally stated

as

$p \vdash_{BL} q$ if and only if $p \vdash_{WL} q$

(Theorem-6)

provided p and q are strongly defined (lists of) formulas: i.e., $\vdash_{BL} DEF_{BL} p$ and $\vdash_{BL} DEF_{BL} q$.

Proof:

First, note that

$\vdash_{BL} e$ if and only if $\vdash_{BL} I[e]$ (*)

provided e is a formula such that $\vdash_{BL} DEF_{BL} [e]$.

$p \vdash_{WL} q$

if and only if $p \models_{WL} q$, by consistency and completeness of WL

if and only if $I[p] \models_{BL} I[q]$, by definition of validity

if and only if $I[p] \vdash_{BL} I[q]$, by consistency and completeness of BL

if and only if $p \vdash_{BL} q$, by (*)

□

3.3. Approximations

In this section, we will show that Weak Logic satisfies Requirement 2 and 3 of the introduction; i.e.,

that proofs on an abstraction level where implementation dependent error situations (such as capacity

constraints in time and space) are disregarded, are valid also on a less abstract level where implementa-

tion errors are considered. As discussed in the introduction, this can be stated by the rule

$$\frac{I \vdash q}{I' \vdash q}$$

(Approximation)

where q must satisfy certain syntactic restrictions.

In order to satisfy Rule of Approximation, non-strict constructs have been defined with care. For example, consider a binary boolean operator op . Each argument can be either defined and true, defined and false, or undefined. For short, say T , F , or U , respectively. The operator op can intuitively be defined by a table of the form:

op	T	F	U
T	aT,T	aT,F	aT,U
F	aF,T	aF,F	aF,U
U	aU,T	aU,F	aU,U

where each a_{ij} is one of T , F , or U .

An implementation dependent error can make an abstractly defined argument undefined; i.e.,

a_{ij} could become aU_j or a_iU or aU,U (i,j in $\{T,F,U\}$)

If a_{ij} becomes U , Rule of Approximation is not violated. In other words, strict functions create no problems, and therefore, user-defined functions can be defined freely. Rule of Approximation is violated if

there are i and j (in $\{T, F, U\}$), such that either

i) a_{ij} is F or U and aU_j , a_iU , or aU,U is T

ii) a_{ij} is T or U and aU_j , a_iU , or aU,U is F

Note that for formulas of form $op(p, q)$ where p and q have no occurrences of the operator op , case (i) is harmless.

The non-strict boolean operators do not violate Rule of Approximation. For example, the non-strict and-operation can be defined by the following table:

$\wedge$	T	T	F	U
T	T	T	F	U
F	F	F	F	F
U	U	U	F	U

Clearly, an implementation error can cause the result U or F , the latter only when the abstract result already was F . Consequently, the non-strict and-operation has no cases of form (i) or (ii) above. Similar

discussions apply to the other non-strict boolean operators, as well as if-expressions. They have no cases of (i) or (ii).

A similar discussion can also be made for quantifiers (even though they are not binary). Universal quantifiers have no case of form (ii), but have cases of form (i), because, in the case where the quantification $\forall x: T[q]$ is false, an implementation error can cause the quantification to become true. (Similarly, existential quantifiers has no (i) case, but has (ii) cases.) As a result, we must restrict Rule of Approximation to formulas q where all universal quantifiers are outmost; i.e., q must have universal form.

Non-strict equality is even more problematic. Consider the intuitive definition below:

$\equiv$	T	F	U
T	T	F	F
F	F	T	F
U	F	F	U/T

Clearly, $\equiv$ has cases of form (i), and even cases of form (ii) if $U \equiv U$ were defined as T. If $U \equiv U$ were defined as $\equiv$, Rule of Approximation is valid for formulas q of form $\forall \dots [not\ q\ 1 \equiv q\ 2]$, provided $\equiv$ does not occur in $q\ 1$ or $q\ 2$. In the case where $U \equiv U$ is true, no such rule is valid, which means that formulas with non-strict equality are only useful within one abstraction level. However, neither of the two versions of non-strict equality is very useful when reasoning about exception handling, as pointed out in section 4.1.

3.3.1. Approximation of Functions

Given two functions f and f' , f' is said to be an approximation of f , denoted $f' \sqsubseteq f$, if their domains have the same base types, say B, and if both

- i) $\forall x: B [x \text{ in } f' \text{ implies } x \text{ in } f]$
- ii) $\forall x: B [f'(x) = f(x)]$

(If one or both functions are total, (i) above can be simplified.) Note that with this definition, it is possible to prove results of the form $f' \sqsubseteq f$ within Weak Logic.

We may now prove that an approximation of f satisfies every theorem about f :

$\vdash qf'(e)$ if $\vdash (qf(e), f' \sqsubseteq f)$ (Theorem-7) provided q has universal form.

With this rule, all occurrences of form $f(\dots)$ in q can be replaced by $f'(\dots)$. Note that occurrences of the form $\dots$ in f will not be replaced by $\dots$ in f' . (One might think that this is satisfied if f' satisfies (i) and every axiom about f , but this is not so, see [Owe 80].)

Proof:

1. $\vdash \forall x:B[x \text{ in } f' \text{ implies } x \text{ in } f]$, by assumption
 2. $\vdash \forall x:B[f'(x) = f(x)]$, by assumption
 3. $\vdash q_f'(e)$, by assumption
 4. $\vdash f'(e) = f(e)$, by 2
 5. $\vdash q_f'(e) = q_f(e)$, by 4 and equality rule since q has universal form
 6. $DEF[q_f'(e)] \vdash DEF[q_f(e)]$, by 5 and since q has universal form
 7. $\vdash q_f'(e)$, by 3, 5, 6, and Modus Ponens.
-

3.3.2. Approximation of Types

Given two types T and T' with the same base type B , T' is said to be an approximation of T , denoted $T' \sqsubseteq T$, if both

- (i) $\forall x:B[x \text{ in } T' \text{ implies } x \text{ in } T]$

- (ii) For each function f associated with T , there is a function f' associated with T' , such that

$$f' \sqsubseteq f.$$

We have here restricted ourselves to types T and T' where T' is more constrained than T . More sophisticated approximations (where T and T' have different base types and where equality on T corresponds to an equivalence relation on T') are described in [Owe 80]. An immediate result is the following version of rule of Approximation:

$$\vdash \forall x:T[q'] \text{ if } (\vdash \forall x:T[q], T' \sqsubseteq T) \quad (\text{Theorem-8})$$

where q is a formula of universal form, and where q' is q with all occurrences of form $f(\dots)$ replaced by $f'(\dots)$. The proof is quite obvious, using Theorem 7. (Note that occurrences of form $\dots$ in T and $\dots$ in f are not replaced by $\dots$ in T' and $\dots$ in f' , respectively.)

Example:

If T' is introduced as a subtype of T by defining a constraint on its values, it is obvious that T' satisfies (i) and (ii) above, and consequently, $T' \sqsubseteq T$. The example below will show how *BoundStack* can be defined as a constrained version of *Stack*:

$$\text{type Stack} \equiv$$

$$\text{total empty} : \rightarrow \text{Stack}$$

$$\text{total push} : \text{Stack*Item} \rightarrow \text{Stack}$$

partial pop : Stack → Stack
partial top : Stack → Item

total length : Stack → Natural -- (including 0)

generators : empty, push -- (defining a "generator basis", see [Dah 78])

def s in pop ≡ s.length > 0
def s in top ≡ s.length > 0

axioms

∀ s : Stack, x : Item

| empty.length = 0 ,

s.push (x).pop = s ,

s.push (x).top = x ,

s.push (x).length = s.length + 1

It is possible to let the domain predicates of *pop* and *top* be implicitly defined, provided the axioms are written as a certain kind of rewrite rules. A discussion of this is beyond the purpose of this paper.

We may now define *BoundedStack* as a constrained version of *Stack* by introducing the constraint:

def s in BoundedStack ≡ s.length ≤ Max

As explained in sections 2.1, *BoundedStack* will inherit (automatically) constrained versions of the

functions associated with *Stack*. By definition of *BoundedStack*, we may conclude that:

emptyBoundedStack ⊆ emptyStack

pushBoundedStack ⊆ pushStack

popBoundedStack ⊆ popStack

topBoundedStack ⊆ topStack

lengthBoundedStack ⊆ lengthStack

and, furthermore, that

BoundedStack ⊆ Stack.

By Rule of Approximation, we may conclude that the *Stack*-axioms are valid for *BoundedStack*:

∀ s : BoundedStack, x : Item

| empty.length = 0 ,

s.push (x).pop = s ,

s.push (x).top = x ,

s.push (x).length = s.length + 1

Furthermore, if we prove a theorem about *Stack*, say

∀ s : Stack | s.pop.length + 1 = s.length ,

we may also conclude

$\forall s: \text{BoundedStack } !s.\text{pop.length} + 1 = s.\text{length}$

However, the theorem

$\text{not } \forall s: \text{Stack } !s.\text{length} \leq 200$

may not be valid for *BoundedStack*; and we may not use Rule of Approximation since the theorem does not have universal form.

□

4. Possible Extensions

We have presented a consistent and complete first order logic, called Weak Logic, which allows partial functions without restrictions. It is a conservative extension of traditional first order logic, which is here represented by Base Logic. The axioms and proof rules of Weak Logic are quite simple, and are similar to those of Base Logic. The semantics of Weak Logic has been chosen to fit our three requirements, which makes Weak Logic specifically suited for:

- i) specification and reasoning about termination aspects of programs
- ii) axiomatization of abstract data types where implementation dependent errors are disregarded
- iii) reasoning about constrained types and partial implementations of abstract data types.

In order to allow implementation dependent errors, non-strict constructs have been defined with care. User-defined functions are considered strict, and the only non-strict constructs are if-expressions, quantifiers, and the non-strict boolean operators (*and then*, *or else*, *implies then*, as well as $\wedge$, $\vee$, $= >$). In particular, there is no non-strict equality function. One disadvantage is that Weak Logic is not well suited for reasoning about exceptions. In this section we shall briefly discuss how Weak Logic can be extended to overcome this problem.

4.1. Exception Handling

Exceptions make it possible to name certain error situations. An unhandled exception will result in an undefined value, whereas a handled exception may result in a defined value. In order to reason about exceptions and exception handling, one needs formulas like

$$\begin{aligned} \forall s:Stack, x:Item [s.full = > s.push(x) \text{ results in } overflow_{stack}] \\ \forall s:Set, x:Item [s.add(x) \text{ results in } overflow_{set} = > s.full] \end{aligned}$$

where $overflow_T$ is an exception associated with type T . The "results in"-predicate must obviously be non-strict, which means that $s.add(x) = overflow_{set}$ does not work since $=$ is strict. We need a non-strict equality, say $\equiv$, such that different error-values are different, for instance,

$$\begin{aligned} not\ overflow_{set} &\equiv underflow_{set}, \quad overflow_{set}, \quad overflow_{set} \equiv overflow_{set}, \\ overflow_{set} &\equiv overflow_{set}.add(x), \quad overflow_{set} \equiv overflow_{set}.has(x) \end{aligned}$$

This non-strict equality function must be introduced as a new kernel operator which is always defined; i.e., $DEF[e1 \equiv e2]$ must be defined as true. A set of axioms should state that it is an equivalence relation, and that it coincides with strict equality for defined expressions. As explained in section 3.3, such an equality function may not be used in premises of Rule of Approximation; however, this is not a major problem since reasoning about exceptions is normally done within a particular abstraction level (as part of a bigger proof).

Exceptions can be introduced in Weak Logic in a fashion similar to "abstract errors" in [Gog 78]. In addition to partial and total functions, we introduce a third kind of functions, called errors. For example, type *Set* could have an associated error

error overflow

which acts like an undefined Set-constant:

partial overflow := *Set*

def in overflow ≡ *False*

except that it is not convenient to let errors have a value of a specific type. In order to formalize error

propagation, errors are considered to be of a universal type. Let us first assume that only one argument is undefined. This allows us to state the following principle:

(propagation-1) $f(\dots, U, \dots) \equiv U$

where *U* denotes an exception and where *f* is a strict function. For instance, if

empty.pop ≡ *underflowstack*, we may conclude that *empty.pop.length* ≡ *underflowstack*

by Law of Propagation. And if we did allow exception handlers in expressions, we might get formulas like

$(\text{empty.pop.length when underflowstack} = > 0) \equiv 0.$

In the case where several arguments are undefined, the following law says that one of them is propagated

but does not specify which one.

(propagation-2) $f(e_1, e_2, \dots, e_n) \equiv e_i$ for some $i = 1, 2, \dots, n$ such that e_i is undefined.

In order to state that exceptions are handled according to name, the following law is necessary:

(uniqueness) *not U* ≡ *U'*

where *U* and *U'* denotes two exceptions with different names.

With these modifications, we believe that Weak Logic can serve as a suitable basis for reasoning

about exceptions as well.

Appendix A: Axioms for Weak Logic

This section presents a set of axioms for Weak Logic, consisting of five general axioms schemas and ten axioms for the type boolean. With these axioms, the Tautology rule of Weak Logic (given in section 2.6) is superfluous, because it can be derived from the other basic proof rules of Weak Logic and the axioms presented here. With the Tautology rule, all the axioms below can be derived as theorems.

General axiom schemas

- (reflexivity) $\forall x : T \mid x = x \mid$
- (symmetry) $\forall x, y : T \mid x = y \text{ implies } y = x \mid$
- (transitivity) $\forall x, y, z : T \mid x = y \text{ implies } (y = z \text{ implies } x = z) \mid$
- (substitutivity) $\forall x, y : T \mid x = y \text{ implies } f(x) = f(y) \mid$
- (if) $\forall x : T \mid q_1^x \text{ then } q_2^x = \text{if } q^x \text{ then } q_2^x$

where T may be any type symbol or product of type symbols. In order to make the last axiom schema closed, the list x should contain all free variables in the equation.

Boolean axioms

- (equality-E) $\forall a, b : \text{Boolean} \mid (a = b) \text{ implies } (a \text{ implies } b) \mid$
- (equality-I) $\forall a, b : \text{Boolean} \mid (a \text{ implies } b) \text{ implies } ((b \text{ implies } a) \text{ implies } (a = b)) \mid$
- (implication) $\forall a, b : \text{Boolean} \mid (a \text{ implies } b) \text{ implies } ((a \text{ implies not } b) \text{ implies not } a) \mid$
- (not) $\forall a : \text{Boolean} \mid \text{not not } a = a \mid$
- (if-1) $\forall a, b, c : \text{Boolean} \mid a \text{ implies } (a \text{ implies if } a \text{ then } b \text{ else } c = b) \mid$
- (if-2) $\forall a, b, c : \text{Boolean} \mid \text{not } a \text{ implies if } a \text{ then } b \text{ else } c = c \mid$
- (or) $\forall a, b : \text{Boolean} \mid (a \text{ or } b) = (\text{not } a \text{ implies } b) \mid$
- (and) $\forall a, b : \text{Boolean} \mid (a \text{ and } b) = \text{not}(\text{not } a \text{ or not } b) \mid$
- (True) True
- (False) $\text{False} = \text{not True}$

The first four axioms are the usual equality axioms, and the if-axiom defines if-expressions in terms of boolean if-expressions. The two first boolean axioms give the relationship between implication and boolean equality. The next two give basic properties of *implies* and *not*. The rest of the axioms define boolean-if, *and*, *or*, *True*, and *False* in terms of previously axiomatized constructs.

References

- [Bar 84] Barringer H., Cheng J.H., Jones C.B.
 "A Logic Covering Undefinedness in Program Proofs"
 To appear in *Acta Informatica*
- [Dah 78] Dahl O.-J.:
 "Can Program Proving Be Made Practical?"
 In *Les Fondements de la Programmation*, INRIA, 1978
- [Dah 85] Dahl O.-J., Langmyhr D.F., Owe O.:
 "ABEL: Language Reference Manual"
 In preparation
- [Fen 83] Fenstad J.E., Normann D.:
 "Algorithms and Logic"
 Lecture notes, Univ. of Oslo, Institute of Mathematics, 1983
- [Gog 78] Goguen J.A.:
 "Abstract Errors for Abstract Data Types"
 In *Formal Description of Programming Concepts*, North Holland, 1978
- [Gor 79] Gordon M.J., Milner R., Wadsworth C.P.:
 "Edinburgh LCF"
 Lecture Notes in Computer Science, Vol 78, Springer-Verlag, 1979
- [Gut 75] Guttag J.V.:
 "The Application to Programming of Abstract Data Types"
 Report CSRG-59 (Ph.D. thesis), Univ. of Toronto, 1975
- [Hoa 69] Hoare C.A.R.:
 "An Axiomatic Approach to Computer Programming"
 CACM 12.10, 1969
- [Kir 82] Kirkerud B.:
 "Undefinedness in Assertion Languages"
 Research Report no. 74, Univ. of Oslo, Institute of Informatics, 1982
- [Knu 70] Knuth D.E., Bendix P.G.:
 "Simple Word Problems in Universal Algebras"
 In *Computational Problems in Abstract Algebra*, Pergamon Press, 1970
- [Luc 84] Lueckham D.C., vonHenke F.W., Krieg-Brueckner B., Owe O.:
 "ANNA: A Language for Annotating Ada Programs"
 CSD Report, Stanford University
- [Owe 80] Owe O.:
 "A Specification Technique with Idealization"
 Research Report no. 50 (Ph.D. thesis), Univ. of Oslo, Institute of Informatics, 1980
- [Owe 82] Owe O.:
 "Program Reasoning Based on a Logic for Partial Functions"
 Univ. of Oslo, Institute of Informatics, 1982
- [Owe 85] Owe O.:
 "A Consistent and Complete Hoare System Based on a First Order Logic for Partial Functions"
 (In preparation)
- [Pra 65] Prawitz D.:
 "Natural Deduction"
 Almqvist & Wiksell, Stockholm 1965
- [Sch 78] Schoenfeld J.R.:
 "Mathematical Logic"
 Addison Wesley 7028, 1967

Acknowledgements:

The author is greatly indebted to Ole-Johan Dahl, for inspiration and for fruitful discussions. I would also like to thank Bill Appelbe, Alma and Dino Karabeg, Viviana Schwarzbein, and Victor Vianu for their comments.

RESEARCH REPORT IN INFORMATICS

"Research report in Informatics" is a series of publications from University of Oslo, Norway, containing new results from computer science, numerical mathematics, software engineering and other parts of "informatics".

The series serves several objectives:

- to provide a faster way of publication than through journals,
- to allow usage of more space than in normal journal publications,
- to be used as study material in seminars, graduate education etc..

All texts will be in english.

The papers will be exchanged with similar productions from research groups elsewhere having corresponding interests.

Order for copies (duplication price) and other correspondence should be sent to:

Institutt for informatikk,
Universitetet i Oslo,
Postboks 1080, Blindern,
OSLO 3
Norway.

No 1	Arne Wang: Generalized Typ's in High-level Programming Languages.	Jan. 1975
No 2	Tom Lyche: Discrete Polynomial Spline Approximation Methods.	Jan. 1975
No 3	Ole-Johan Dahl and Arne Jonassen: Analysis of an Algorithm for Priority Queue Administration.	Jan. 1975
No 4	Olav Dahl: On the Problem of Solving Linear Algebraic Equations Associated with Matrices that are Polynomial Functions of a Square Matrix.	Jan. 1975
No 5	Tom Lyche: Discrete Cubic Spline Interpolation	Feb. 1975
No 6	Arne Wang: An Axiomatic Basis for Proving Total Correctness of Goto-Programs.	April 1975
No 7	Tom Lyche: Computation of B-Spline Gram Matrices.	June 1975
No 8	Arne Jonassen: Additional Notes on the Normal P-tree Forest.	Sept. 1975
No 9	Arne Wang: Correctness of Transformations of Recursion to Iteration in Programs.	Oct. 1975
No 10	Arne Wang: The Semantics of Programs. Weakest Preconditions Revisited.	Nov. 1975
No 11	Tom Lyche: A Note on the Condition Numbers of the B-spline Bases	June 1976
No 12	Olav Dahl: Numerical Solution of Partial Differential Equations on Parallel Computers. A case study.	Nov. 1976
No 13	Ole-Johan Dahl: An Approach to Correctness Proof of Semiroutines.	Jan. 1977
No 14	Arne Jonassen: Priority Queue Processes with Biased Insertion of Exponentially Distributed Input Keys.	May 1977
No 15	Arne Jonassen: A Formal Description of Data Table Processes.	May 1977

- No 16 Ragnar Winther:
Iterative Methods for a Class of Linear Equations with Application to Galerkin Approximations of a Parabolic Control Problem.
- No 17 Stein Gjessing:
Compile Time Preparations for Run Time Scheduling in Monitors.
- No 18 T. Lyche og R. Winther:
A Stable Recurrence Relation for Trigonometric B-splines.
- No 19 Arne Jonassen:
A Study on the Reducibility of the Polynomials $x^m + x^{2p}$.
- No 20 Reiji Nakajima:
Proving Abstract Specifications to be Valid for their Program Implementations.
- x) No 21 H. T. Faye-Schjöl, O. Hesjedal, D. F. Langmyhr, O. Owe:
The Programming and Specification Language ABEL.
- No 22 Reiji Nakajima:
Type-Class of Types or Partial Types - for Program Specification Structuring; and their Formalization a First Order Logic.
- No 23 Arne Jonassen:
Analysis of the "Dutch Flag Problem".
- No 24 Arne Jonassen:
Special Priority Queues with General Incident Sequences.
- x) No 25 Ellen Hisdal:
Conditional Possibilities, Independence and Noninteraction.
- No 26 Olaf Owe:
Notes on Partial Correctness.
- No 27 Arne Jonassen:
Alternative Solution to the "Dutch National Flag" Problem.
- No 28 Tom Lyche and L.L. Schumaker:
An ALGOL Package for Computing Smoothing and Interpolating Spline Functions.
- No 29 Stein Gjessing:
Monitors with Associated Processors
- No 30 Søren Øivind Wille:
Finite Element Formulations of Navier-Stokes Equations.
- No 31 Stein Gjessing:
The Structure of the MUTE Multi-Microcomputer System.
- x Not published

May 1977

June 1977

Sept. 1977

Sept. 1977

Nov. 1977

Nov. 1977

Nov. 1977

Nov. 1977

Nov. 1977

Dec. 1977

Dec. 1977

Jan. 1978

Jan. 1978

Feb. 1978

March 1978

March 1978

- No 32 Vesko Marinov:
Maximal Clause Length Problem.

April 1978

- No 33 Ole-Johan Dahl:
Can Program Proving be Made Practical?

May 1978

- No 34 Arne Jonassen:
The Limiting Behaviour of P-trees with Incident Sequence $O(n^m d^n)$

August 1978

- No 35 Arne Jonassen:
The "Chain Gang Schedule", Theoretical Solution.

August 1978

- No 36 Arne Jonassen:
The Polynomials $x^m + x^{2p}$, A new Approach to the Question of Irreducibility.

August 1978

- No 37 Tom Lyche:
A Newton form for trigonometric Hermite interpolation.

Sept. 1978

- No 38 Dag F. Langmyhr and Olaf Owe:
Revised Report on the Programming and Specification Language ABEL.

Oct. 1978

- No 39 Ellen Hisdal:
Acquisition of Information in a Learning Process.

March 1977

- No 40 T. Dokken and T. Lyche:
A Divided Difference Formula for the Error in Numerical Differentiation.

Dec. 1978

- No 41 Arne Jonassen, Per-Arne Rindalsholt, Rolf Sædahl and Gunnar Valsø:
A study on some Pseudo-Random Generators for a 36-bit Computer.

Dec. 1978

- No 42 Ellen Hisdal:
Conditional and Joint Possibilities of Higher Order.

July 1978

- No 43 Ellen Hisdal:
Concrete and Mathematical Sets, Measures of Second Order Possibilities.

Aug. 1978

- No 44 Ellen Hisdal:
Particularization - The Theory of Fuzzy Sets versus Classical Theories.

Nov. 1978

- No 45 Stein Gjessing:
Monitors with arrays of condition variables and proof rules handling local quantities.

June 1979

- No 46 Stein Gjessing:
Microcomputer Interrupt Servicing in an Environment of Processes and Monitors.

June 1979

- No 47 Stein Gjessing:
Microcomputer Software Design and Programming using the Concepts of Processes and Monitors - a Case Study.

June 1979

No 48	Ole-Johan Dahl: Time Sequences as a tool for describing program behaviour	Aug. 1979
No 49	Olav Dahl: On the choice of pivot elements in Gaussian elimination	March 1980
No 50	Olaf Ove: A specification Technique with Idealization	March 1980
No 51	Ellen Hissdal: The IF THEN ELSE Statement and Interval-Valued Fuzzy Sets of Higher Type	May 1980
No 52	Per M.A. Bothner: High-level data types in programming languages	Oct. 1980
No 53	Neelam Soundararajan: Semantics of communicating sequential processes	Febr. 1981
No 54	Robert W. Gundersen: Choosing the r-dimension for the FVC family of clustering algorithms	March 1981
No 55	Vesko Marinov: Resolution theorem prover in a program verification environment	May 1981
No 56	Neelam Soundararajan: Axiomatic semantics of communicating sequential processes	May 1981
No 57	Neelam Soundararajan: A note on the Egli-Wilner order and Smyth's problem	May 1981
No 58	Neelam Soundararajan: Consistency and completeness of a proof system for CSP	June 1981
No 59	Neelam Soundararajan: Proofs of correctness of CSP programs	June 1981
No 60	Arne Wang: Denotational and Axiomatic Semantics for Non- deterministic Goto-programs	Aug. 1981
No 61	Bjørn Kierkud: The Semantics of Programming Languages. Vol. 1 & 2	Aug. 1981
No 62	Inge Henningsen og Knut Liestøl: The behaviour of a nerve cell model with strong inhibitory postsynaptic potentials	Sept. 1981
No 63	Ellen Hissdal: Sets and natural language	Sept. 1981
No 64	Ellen Hissdal: Possibilities and probabilities	Sept. 1981

No 65	Olaf Ove: A Note on Qualifier Removal	Sept. 1981
No 66	Neelam Soundararajan and Ole-Johan Dahl: Partial Correctness Semantics of Communicating Sequential Processes	Febr. 1982
No 67	Rolf Nossum: Algebraic simplification in the prover system	April 1982
No 68	Per Erik Koch: Collocation by L-Splines at Gaussian Points	April 1982
No 69	Dag F. Langmyhr: A General Assembler for Micro-processors	May 1982
No 70	Arne Wang: Coroutine sequencing in Simula. Part I, II og III	Aug. 1982
No 71	Bjørn Kierkud: Completeness of Hoare-Calculi revisited	Aug. 1982
No 72	Per Erik Koch: Jackson theorems for generalized polynomials with special applications to trigonometric and hyperbolic functions	Oct. 1982
No 73	Stein Krogdahl: Protection in Simula: A new proposal	Nov. 1982
No 74	Bjørn Kierkud: Undefinedness in assertion languages	Nov. 1982
No 75	Per Erik Koch: Collocation by L-splines at transformed Gaussian points	Jan. 1983
No 76	Rolf Nossum: A proof technique based on simplification	May 1983
No 77	Stein Gjessing: Verification of Monitors based on a Partial Correctness Semantics	July 1983
No 78	Per Erik Koch: Error Bounds for interpolation by fourth order trigonometric splines.	Sept. 1983
No 79	Ole-Johan Dahl: Notes on a LIFO Disciplined Simplex Algorithm	Febr. 1984
No 80	Elaine Cohen Tom Lyche Richard Riesenfeld: Discrete box splines and refinement algorithms	Des. 1983
No 81	Tom Lyche: A note on Chebyshevian B-splines	April 1984
No 82	Per Erik Koch: Exponentially fitted collocation methods for singularly perturbed two-point boundary value problems.	Mai 1984

No.	83	Stein Krogdahl An efficient implementation of simula classes with multiple prefixing	June 1984
No.	84	Ole-Johan Dahl: LOGIC OF PROGRAMMING and SPECIFICATION	July 1984
No.	85	Per Erik Koch: MULTIVARIATE TRIGONOMETRIC B-SPLINES	July 1984
No.	86	Rolf Nossens Decision Algorithms for Program Verification	July 1984
No.	87	T.Lyche and K.Mørken: Making the Oslo Algorithm More Efficient	Aug. 1984
No.	88	E.Cohen, T.Lyche and K.Mørken: Knot Line Refinement Algorithms for Tensor Product B-spline Surfaces	Dec. 1984
No.	89	Olaf Owe: An Approach to Program Reasoning Based on a First Order Logic for Partial Functions.	Febr. 1985