A Generator for Type Checkers

Dissertation
der Fakultät für Informations- und Kognitionswissenschaften
der Eberhard-Karls-Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von

Dipl.-Inform. Holger Gast
aus Moers

Tübingen
2004

Tag der mündlichen Qualifikation:	8. Juni 2005
Dekan:	Prof. Dr. Michael Diehl
1. Berichterstatter:	Prof. Dr. Rüdiger Loos
2. Berichterstatter:	Prof. Dr. Herbert Klaeren
3. Berichterstatter:	Prof. Dr. David R. Musser
	(Rensselaer Polytechnic Institute, NY, USA)

Abstract

Compiler-compilers are tools that generate substitutes for hand-written compiler components from high-level formal specifications. Such tools exist for lexical, syntactic and semantic analysis, optimizers and code generation. The established benefits are reduced development time and increased confidence in the correctness of the resulting software.

This thesis presents a generator for type checkers. Given a description of the type system by typing rules, the generator yields a type checker that constructs proofs using the typing rules. Unlike earlier approaches, we derive suitable notions of *proof* and *typing rule* from an analysis of type systems and from corresponding constructs in mathematical proof theory. The approach thus respects the structure and intention of the typing rules, rather than expressing the rules in some pre-existing formalism.

The given applications comprise type checkers for imperative, object-oriented and functional languages, including ML type inference. The typing rules for these checkers directly represent those found in the literature. They naturally describe the typing of single language constructs and they can be re-used in different checkers.

We use the generator to develop the language SAGA for generic programming. Generic programming has become a standard approach to creating reusable and reliable software, particularly through the wide-spread use of the C^{++} Standard Template Library (STL). Existing C^{++} compilers cannot type-check generic algorithms before instances are generated, hence errors manifest themselves only when the algorithms are used. SAGA overcomes this problem by a novel language design that enables generic algorithms as found in the C^{++} STL to be type-checked such that the correctness requirements stated in algorithm interfaces are obeyed and instantiation never fails. It therefore turns the aims of the earlier proposal SUCHTHAT into a concrete language design.

Zuammenfassung

Compiler-compiler generieren aus formalen Spezifikationen Komponenten für Compiler, um dort handgeschriebenen Code ersetzen. Solche Generatoren existieren für die lexikalische, syntaktische und semantische Anlayse, für Optimierer und die Coderzeugung. Es hat sich gezeigt, daß die Entwicklungszeit abnimmt und gleichzeitig das Vertrauen in die Korrektheit der Software steigt.

Die vorliegende Dissertation beschreibt einen Generator für Typchecker. Er erzeugt aus einer Spezifikation eines Typsystems, die in Form vom Typregeln gegeben ist, einen ablauffähigen Typchecker, der Typherleitungen mit Hilfe der gegebenen Regeln konstruiert. Abweichend von früheren Vorschlägen werden passende Definitionen für *Typherleitung* und *Typregel* durch Analyse existierender Typsysteme und der mathematischen Beweistheorie gewonnen. Auf diese Weise reflektiert der Ansatz die Struktur und Intention der Typsysteme, anstatt die Typregeln in einem bereits vorhandenen Formalismus auszudrücken.

Als Anwendungen werden Typchecker für imperative, objekt-orientierte und funktionale Sprachen, einschließlich der ML Typinferenz, formalisiert. Die Typregeln dieser Checker korrespondieren direkt mit den aus der Literatur bekannten. Da sie sich einzelnen Sprachkonstrukten zuordnen lassen, können sie in verschiedenen Checkern wiederverwendet werden.

Eine spezielle Anwendung ist die Sprache SAGA für die Generische Programmierung. Die Generische Programmierung ist zu einem Standardansatz zur Erstellung verläßlicher und wiederverwendbarer Software geworden, insbesondere durch die weite Verbreitung der C^{++} Standard Template Library (STL). Der C^{++} Compiler kann allerdings die generischen Algorithmen erst dann prüfen, wenn konkrete Instanzen generiert werden. Fehler in den Algorithmen manifestieren sich daher erst bei der Benutzung. SAGA löst dieses Problem durch eine neues Sprachdesign, das es erlaubt, generische Algorithmen der STL so zu überprüfen, daß die deklarierten Korrektheitsbedingungen erfüllt sind und die Instanzgenerierung nie fehlschlägt, wenn der Typchecker die Algorithmen akzeptiert. Damit realisiert SAGA die Ziele des früheren Sprachvorschlags SUCHTHAT in einem konkreten Sprachdesign.

Danksagungen

Zuallererst danke ich meinem Doktorvater, Prof. Dr. Rüdiger Loos, herzlich für die stetige Unterstützung meiner Arbeit. Durch die Mitarbeit in seiner Computeralgebra-Gruppe während meines Studiums wurde ich erstmals auf die generische Programmierung und auf die damit verbundenen Typfragen aufmerksam. Sein Interesse an meinen Ergebnissen und Fragestellungen hat mich immer wieder beflügelt und vor neue Herausforderungen gestellt.

Prof. Dr. Herbert Klaeren gab mir die Möglichkeit, die vorliegende Fassung als Mitglied seiner Arbeitsgruppe Programmiersprachen und Übersetzer abzuschließen. Dieser erneuten Überarbeitung verdankt die Dissertation viel in Hinblick auf den direkten Ausdruck der Typregeln und die effiziente Implementierung.

Weiterhin danke ich meinen Kollegen am Wilhelm-Schickard-Institut. Dr. Roland Weiss und Dr. Volker Simonis haben viel zu meinem Verständnis der Sprache C^{++} beigetragen. Dr. habil. Christoph Schwarzweller hat die Arbeiten an der Sprache Saga begleitet und war stets offen für Diskussionen über deren Typsystem. Dr. Carsten Sinz hat Teile von Kapitel 2 durchgesehen.

Meiner Familie, insbesondere meinen Eltern Werner und Sibille Gast, danke ich herzlich für den aufmunternden Rückhalt und die Förderung meines frühen Interesses an der Programmierung. Ich danke Dorothea Flothow für die Unterstützung in der Endphase der Überarbeitung.

Contents

Chapter 1

Introduction

Type systems are essential to many modern programming languages and type checkers are standard components of their compilers. Like lexers, parsers and code generators, type checkers must be specified and implemented accurately by the compiler writer. Unlike these other components, they do not enjoy any specific tool support: There is no accepted notion of a type check generator.

In this thesis, a *type check generator* is understood to be a program that takes some description of a programming language's type system as input and generates a type checker to be used in the semantic analysis phase of the language's compiler (Figure 1.1, adapted from [Muc97]).

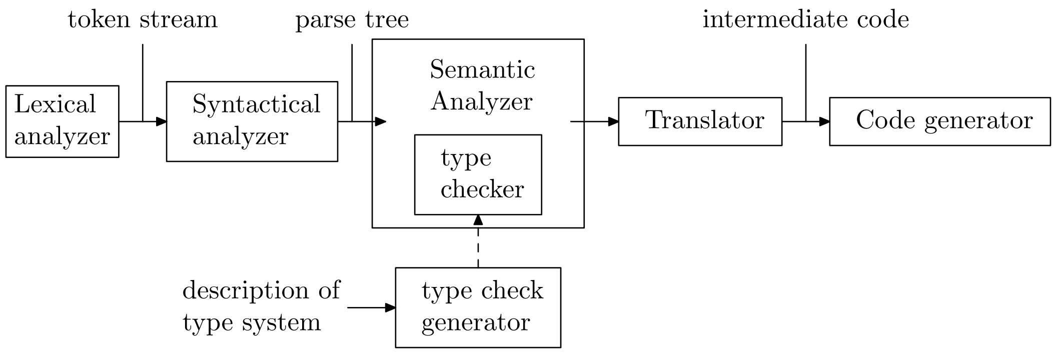


Figure 1.1: The Generated Type Checker in a Compiler

I will use the term *type checker* regardless of the actual task that the component performs: For a particular language, it may be sufficient to check that the type annotations given by the programmer are consistent; for a different language, the type checker may perform type inference (type reconstruction). For most languages, the type checker lies inbetween these extremes: Even in languages with program annotations, at least the result types of expressions must usually be reconstructed; in languages with type inference, the results may be matched against an optional function or module signature provided by the programmer. The perspective on type systems must therefore be biased slightly towards type inference.

In this thesis, the emphasis in semantic analysis is on type checking. Hence, the implementation of the type check generator adapts the compiler structure, such that the language's syntax is specified along with its type system (Figure 1.2). This approach allows for a tight integration of parser and type checker and facilitates prototyping of new languages. A limited amount of translation is available by rewriting; the result of translation is output textually.

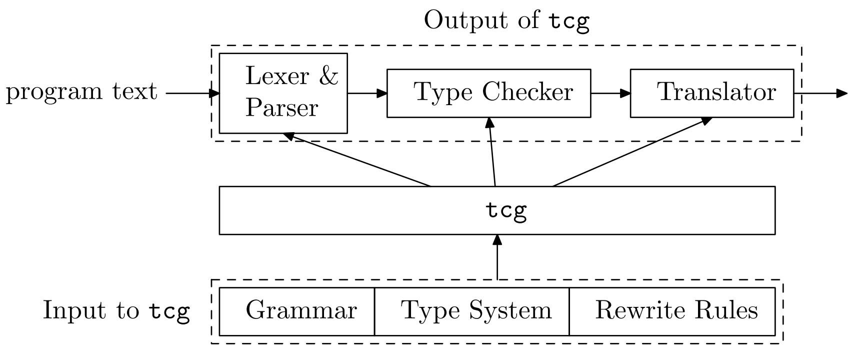


Figure 1.2: Emphasis on Type Checking

1.1 Statement of the Thesis

From a software-engineering point of view [CHW98], in order to design a type check generator it is necessary to find a common abstraction of existing type systems, and then to implement that abstraction. Four essential requirements must be fulfilled by the chosen abstraction to match the situation outlined in Figure 1.2:

1. The abstraction must capture the entire process of type checking a raw parse tree.
2. The abstraction must be uniform, that is a specific type checker must be obtained by (mechanically) instantiating the abstraction. This is contrary to software-engineering, where the construction of instances may (and will) involve human activity, for instance by implementing a design pattern [GHJV95].
3. The abstraction must be stated formally: Type systems deal with programs as recursive objects, which necessitates strong invariants. Furthermore, the semantics of compilation may depend on the type check, for instance for overloaded identifiers [GR80, Cor82, Str97, WB89, HHJW96, Jon99] and conversions [BTCGS91]. At least in principle, the generated type checker must be verifiable by reasoning about the type system's specification (assuming that the generator is implemented correctly).
4. Despite the formality, the type checker's behaviour must also have an intuitive description: The straightforward portions of type system should be expressible without going through a formal process. Furthermore, it should be possible to prototype a type system [Rep84, BS86, BCD$^+$89, LP03].

Several abstractions of type systems have been proposed (see Section 5.1), but none of them fulfills all of the above requirements. In particular (2) and (4) are not implemented satisfactorily. Consequently, the available tools mostly require an insight into the workings of type checkers in order to describe a type system. This failure seems to be mainly due to the fact that type checkers are expressed in some pre-existing formalism, rather than in a formalism tailored to type systems. I therefore suggest that a new abstraction is needed and it should be designed with as little prejudice as possible. The following, fifth, requirement captures this intention:

5. The abstraction should result from an analysis of the common structure of type systems.[1]

Motivation of the Thesis There is vast diversity of type systems with different aims, points of emphasis and application areas. However, the systems exhibit a common tendency to express *implications* between typing judgments, which is made explicit in their presentation by deduction (or inference) rules.

For concreteness, consider the simply-typed λ-calculus with its basic constructs of variables, functions and function applications [Mit90]:

$$\frac{\Gamma \vdash f : s \to t \quad \Gamma \vdash e : s}{\Gamma \vdash (f\,e) : t}(\text{apply}) \qquad \frac{\Gamma_x \cup \{x : s\} \vdash e : t}{\Gamma \vdash \lambda x.e : s \to t}(\text{abs}) \qquad \frac{x : s \in \Gamma}{\Gamma \vdash x : s}(\text{var})$$

To type-check an application $(f\,e)$, the function f and argument e are checked independently, and *if* they agree on the actual and expected argument type, *then* the function application can be performed. Similarly for a function $\lambda x.e$, *if* the type check of e succeeds under the assumption that the argument type is s, *then* the expression will have (function-) type $s \to t$. The last rule states that the type assumptions are used for variables. Any typing of some expression e must be obtained by constructing a derivation (or proof) by these three rules.

The form of presentation by deduction rules is ubiquitous. This observation alone would justify using rules as an abstraction according to the practice of domain analysis [Cop98]. Furthermore, it appears that the commonality in formulation is not a coincidence: Type systems approximate the meaning of programs, they derive true propositions about the behaviour at run-time.[2] Hence, the main mode of inference is to constructively deduce true propositions from other true propositions, possibly using assumptions. Therefore, even if a new style of presentation were to emerge, its fundamental concept of type checking would likely be implication as well.

Statement of the Thesis

I propose *type-checking-as-proof-search* as a suitable abstraction for designing a type check generator: Type systems can be understood and formalized as logical systems such that there is a typing derivation if and only if there is a proof in the logical system. Moreover, the description of the type system will be natural (relative to the specification by deduction rules), and proof search can be implemented effectively.

[1]I am aware of the interaction between abstractions and structure, but hope that the fifth requirement nevertheless clarifies the intention as stated above.

[2]Pierce [Pie02] speaks of “lightweight formal methods”.

1.1.1 Remark. In this general form, this statement is complementary to the well-known Curry-Howard isomorphism phrased as *formulae-as-types* (or *propositions-as-types*) (e.g. [How80, CH88, Bar91, TS00]): There, proof search starts with a type (as an encoding of a formula to be proven) and the aim is to find a λ-term with the type. That term is then a direct encoding of the found proof and type checking means proof checking. In the above thesis, on the contrary, proof search is the means to find a type for a given term. Furthermore type checking for programming languages cannot be straightforwardly reduced to type checking for higher-order logics; this is true in particular for type inference (see Sections 1.2.3.5 and 5.3.2).

1.1.2 Remark. Deduction rules in the literature serve as *specifications* of type systems, they are not expected to convey the operational semantics of the type checker. This distinction is particularly important where type checkers perform type inference, possibly with computations on hand-crafted data structures, in order to determine the correct instantiations of (meta-)variables in the rules. In such cases, it will be necessary to re-factor the deduction rules to make them suitable to proof search. This step may at first seem to be an immediate disadvantage of the proposed approach over conventional typing algorithms. However, in the examples analyzed in Chapter 4 (also Sections 2.4.1.2, 5.3.2), the re-formulation reveals new relations between type systems and yields a new conception of their structure.

Discussion The choice of proof search as an abstraction for type checking has some immediate consequences, which I will discuss briefly now. A more thorough motivation of the thesis through analysis of existing type systems can be found in Section 1.2.

As the first consequence of the proposed abstraction, type checking is a completely symbolic process. Compare this to the conventional approach, where deduction rules are used for specifying a type system, while the type checker is a general algorithm with general data structures. For most type systems of programming languages, that generality is not needed, as types are terms, and the basic operations only decompose them and substitute variables. We can also observe that most type systems of conventional imperative languages have even simpler notions of "type", because they compare types by name rather than by structure. Nevertheless, some type checkers employ more generality. For instance, recursive types are often represented as cyclic graphs [AHS86, AC93, Wri94]. Although the type check can be done symbolically [BH97], there remains a discrepancy.

Next, the choice of proof search establishes a direct connection to the well-developed field of automatic theorem proving: The type check generator can re-use the techniques employed there. It turns out during the study of existing tools (Chapter 5) that a simple adaptation of a theorem prover will not be sufficient. The proof structure needed to express type checkers (Section 1.2, Chapter 2) is akin to proof search with Hereditary Harrop Formulae [MNPS91] (or higher-order logic programming [Mil91, NM98], theorem proving in higher-order logic [Pau86, Pau94], Section 5.3.2) rather than first-order techniques ([Rob65, Gal86], Prolog [And92]).

Finally, the generated type checkers are sound by construction (assuming that the generator is implemented correctly): Every successful type check explicitly (Chapter 3) constructs a proof that can be interpreted as a type deduction. This property is a distinct advantage in prototyping type systems for new languages.

Related Work Related work needs to be considered in three directions:

1. The proposed abstraction over type systems can be compared to other possible abstractions. Abstractions based on constraints (Section 5.2) have recently attracted interest [OSW99, Sul00, SS01, Sul01, AF02, AF04]. The central idea is to separate type checking into two phases, *constraint generation* and *constraint solution*. It is expected that constraint generation is a straightforward recursion over the parse tree, while the main part of the type checker focuses on the solution of constraints. Constraints in this sense arise naturally in a variety of type systems, foremost perhaps those including subtyping [Mit91, EST95b, JP99, AWL94, Pot01]. A novel line of research [SS01, SSW04] has come from the application of constraint handling rules to constraint solution. TCG is complementary to this line of research, as it considers the construction of the type derivation, which is implicit in the recursive traversal for constraint generation. If desired, TCG's deferred goals (Section 2.3) can be interpreted as unsolved constraints (Section 6.2.3).
2. The resulting type check generator can be compared to other tools used for expressing type checkers. This will be done in Section 5.1. In this direction, research began in the early 1980s, and concentrated on formal descriptions of programming languages, with the double goals of generating programming environments and reasoning formally about the specification (and implementation). The programming environments included (incremental) type checkers, mostly for Pascal-like, imperative languages [Des84, CDDK86, BCD$^+$89, TR81, RT88, Deu91, TD01]. However, in this period the aim was to express type checkers in a given formalism, rather than devising a formalism for type checking. The specific features, for example polymorphic let, had to be handled by mechanisms outside the scope of the formalism [Des84, CDDK86].
3. The type check generator processes a description of a type system that is conceived as a logical system, and it implements general mechanisms for dealing with that system. This intention is related to the area of logical frameworks [Pfe01]. A comparison is given in Section 5.3. In a case study Section 5.3.2 shows how the Hindley-Milner type system for ML [Mil78, DM82] can be expressed in ISABELLE [Pau94].

Contributions The main contribution of this thesis is a novel understanding of the tasks of type checking and type inference.

- I design a formalism to capture the specifications of type systems (Section 1.2, 3.1).
- The design (Section 1.2) exhibits a strong structural relation of type systems to conventional proof-theoretic constructions in logic. This relation explains several features, foremost the ML-style generalization (Section 1.2.3.5) and constraint-based type systems (Section 1.2.3.6).
- The formalism can be given an operational semantics for proof search (Chapter 2).

Towards the implementation of type checkers for compilers, four results have been attained:

- I implement a type check generator (Chapter 3).
- The main features of several widely employed type systems have been formalized (Chapter 4).
- That formalization exhibits many commonalities among the systems, such that typing rules valid in one system can re-used in new systems. This result facilitates prototyping of new programming languages [LP03].
- The type check generator can be understood as the missing generalization of constraint generation for constraint-based type inferences (Sections 1.2.3.7, 1.2.3.8, 6.2.2). It thus complements recent developments with constraint handling rules [SS01].

Towards language support for generic programming

- I present in Section 4.5 the language design and type checker of SAGA, which refines and implements Schwarzweller's [Sch02, Sch03] proposal to use signatures and adjectives to describe the interface of generic algorithms [MS94, Sch96a, Sch96b].

Finally, there is a minor contribution to compiler implementation in logic programming with higher-order abstract syntax [PE88].

- The study of the Hindley-Milner system in Section 5.3.2 shows that the classical type inference Algorithm W can be implemented directly in ISABELLE . This result solves a long-standing question [Pfe88, DP91, Lia97, Han98, Lia02].

Scope Before we embark on the investigation, it must be made explicit that type systems are too vast an area for capturing every proposed system within a single framework.

- The type systems' motivations range from mathematical logics [ML84, CH88, Hin69, BG00] to the necessity to distinguish data types for compilation [KR88, Rit93, Str97]. In between these extremes, there is room for compiler optimizations [AWL94, Wri94, Ler98], static analysis [NNH99, Chapter 5], software-design methodologies [Boo91, SOM93, GJS00, BCK$^+$01], and safety-considerations on executable code [MWCG99, Nec97].
- They encompass the tasks of checking the consistency of type annotations [JW85, Wir88, Str97], to reconstructing the type of every expression and function in a program [Mil78, DM82, WB89].
- They embrace programming language features from numeric calculation, over module systems [HL94, Ler00], dynamic dispatch for object-oriented programming [Str97, AC96, GJS00, EST95b, BSG03, OW97, AFM97], and parameterization [Mil78, AC96, Str97, AFM97] to algebraic considerations behind the program text [JS92, Web93, San95, Sch96a].
- Their descriptions take the form of natural language standards documents [JW85, ISO98], deduction systems [DM82, CDDK86, Bar91], algorithms [Mil78] and constraint systems [Wan87, Pot01, OSW99, AFFS98, EST95b].

Even this very brief sketch indicates that the scope of the type check generator needs restriction. For an analogy, the established parser generators do not support every language syntax imaginable, but they provide a framework that can be adapted conveniently to a certain class of syntax constructs commonly found in programming languages.

In designing the type check generator (Section 1.2) I will therefore start out from the simply typed lambda calculus, gradually adding language constructs to keep the design self-contained and well-defined. Section 4.1 retraces these steps with the resulting system.

I explicitly exclude from the investigation systems with dependent types [XP99, Aug98], higher-order polymorphism [Bar91, OL96, Car93] and type systems that serve to represent logics via the Curry-Howard isomorphism [How80, CH88, Bar91] (except for those fragments found in programming languages). Although, for example the Calculus of Construction version with β-normal types [CH88, Section 6.1] seems amenable to a treatment in Tcg, the area introduces too many further questions.

1.1.1 Overview

The remainder of this introductory Chapter is organized as follows: Section 1.2 introduces the design of Tcg by motivating examples taken from programming languages. It is organized around the typing constructs of languages and justifies the choice of *type-checking-as-proof-search* as the key abstraction.

Chapter 2 gives an operational semantics to the proof formalism proposed in Section 1.2, in essentially the same manner that operational semantics can be assigned to Prolog programs [And92].

Chapter 3 describes the implemented interpreter for Tcg. In particular, I show how the proof structure of Chapter 2 can be implemented efficiently.

Chapter 4 applies the constructed tool to an extensive number of examples. Starting from the basic λ-calculi with polymorphism, I proceed to imperative and object-oriented languages.

Chapter 6 concludes with a summary and future directions of research.

1.2 Designing a Type Check Generator

Deduction rules in the literature on type systems are meta-level objects: They serve to formalize a type system, but they are not formalized themselves. In order to make proof search a vehicle for type checking, we must fix the form of deduction rules. In a more general sense, the aim of this section is thus to establish a proof theory for type systems. Several possible approaches can be conceived:

1. Foundational studies on the intention of type checking.
   Starting from the semantics of type judgments, it may be possible to give a complete axiomatization. This approach parallels the basic goal of giving complete deduction systems for logics (e.g. [Gal86]). It requires a thorough analysis of the commonalities of the semantic domains, i.e. the meaning of programs, that type systems approximate. Although much of the existing literature on type systems is concerned with axiomatizing semantics, it does not intend to establish a *common* system for all applications.

2. Encoding type systems as logics.
   The type judgments can be expressed as predicates in a suitable logic: The $\vdash$-relation employed in many type systems would be a ternary predicate, and context, expressions and types would be logical terms. The work on TYPOL [BCD+89] (Section 5.1.4) follows this trail. It has the disadvantage that no support for type systems is available *a priori*: Every predicate needs to be axiomatized. This restriction can also be seen as an advantage: The designer of the type system is forced to make explicit all structural properties about contexts, variables and lookup of type assumptions. However, the descriptions of type systems resemble type checkers written in a logical programming language.
3. Abstraction over existing rules.
   By regarding proposed typing rules as objects in their own right, one can identify recurring patterns. The advantage of this approach over those previously considered is the direct connection to the type systems' specifications: If all the rules of the specification can be expressed as an input to the type check generator, then the resulting typings will trivially obey the specification.

I will follow last approach in this section. At the center of the study is the type system for MiniML [CDDK86], which will eventually be implemented in Tcg in Section 4.1. With each added construct, I will discuss related issues in the literature and their possible integration into Tcg.

I will also briefly outline the relation to logical systems according to the above approach 2 in Section 1.2.2. The comparison focuses on the relation between natural deduction and sequent style formulations. It is based on the textbooks by Troelstra and Schwichtenberg [TS00], Negri and von Plato [NP01], and Gallier [Gal86], and on the classical presentations by Gentzen [Gen35] and Prawitz [Pra65].

1.2.1 Preliminary Considerations

Before we can explore the commonalities of type systems, I establish the common background against which the examples are set. (A strictly inductive procedure would be less prejudiced (Section 1.1), but also far less directed and probably less readable.)

1.2.1.1 Notation

In citing material from diverse fields of type literature, we face the problem of varying notation. Obviously, there are two alternative solutions that can be adopted: Either cite all material with the exact notation used in the original paper, or transliterate the material to a common notation.

I have decided for the latter alternative for two reasons: First, the overall presentation will be more coherent. I wish to emphasize commonalities, without being disturbed by superficial variabilities of notation. Second, the changes to be made are minor, they consist in the replacement of symbols for the most part. I will use the following naming convention.

Symbol	Meaning
$\equiv$	syntactic equivalence
$=$	equality
$\vdash$	derivability
$s, t, r, \ldots$	Types and type schemes
$\alpha, \beta, \gamma, \delta$	Type variables
σ, τ	Substitutions
Γ	Context (type assumptions)
e, f, g	Expressions
x, y, z	Identifiers (variables) in expressions

In those rare cases where the structure of the presented material must be modified in non-trivial ways to exhibit a commonality, I include the original material verbatim in a footnote.

1.2.1.2 Levels of Reasoning

In the subsequent discussion, we will frequently have to distinguish several levels (or layers) of discourse: Some proof system, language, notation etc. will be used to describe a different proof system, language, notation, etc. Following established usage, I will refer to the described system as the *object-level* system (or *object* for short), and the system employed for the description as the *meta-level* system.

This terminology suggests that there are but two levels involved in the discourse. This is in general not the case: On the contrary, the *reduction* of one problem to another problem is an ubiquitous technique in computer science. (Not only in theoretical computer science, but even down to the very practical fields of modules and abstract data types, e.g. [Hoa72].) Several reductions must usually be applied in a chain to obtain the solution, and each reduction introduces a new level of reasoning about the correctness of the final solution. I will therefore extend the notions of *meta-level* and *object-level* to each pair of consecutive levels.

1.2.1.3 Typing Judgments and Derivations

We study type systems formulated by deduction rules. Consequently, the notation and terminology will be taken from proof theory [NP01, TS00, Gal86]. Throughout the text, I will use the prefix *typing-* to emphasize a relation to type systems, as opposed to logics or the type check generator.

Proof systems relate *judgments* (or *assertions*), which may be for instance formulae, propositions, or sequents. In this section, the exact form of judgments will be determined by the type system in question.

1.2.1 REMARK. In logic, it is customary to distinguish between assertions and propositions [NP01]: A proposition is an object under consideration, while the assertion, that the proposition holds, belongs to the meta-level. Consequently, a proposition P must be distinguished from the assertion $\vdash P$ (read as "P is derivable"). Deductions (see below) are hence labeled with assertions $\vdash P$, which is abbreviated as a label P.

A *rule* consists of a sequence of *premises* $P_1 \mathinner{..} P_n$ and conclusion C. (The term "premises" is often spelled *premisses* [NP01]). Each of the P_i and C is a judgement. The rule is written with a horizontal line:

$$\frac{P_1 \mathinner{..} P_n}{C}$$

A rule without premises ($n = 0$) is also called an *axiom.* I will also write *deduction rule* to distinguish this form of rule from others, for instance rewrite rules.

The judgments of rules may contain *meta-variables* which designate an arbitrary object of a given class. A *rule instance* is a rule in which meta-variables have been replaced by concrete objects. A rule that contains meta-variables is also called a *rule schema.*

Given a fixed set of rules, a *deduction* (or *derivation*) is a (finite) tree labeled with judgments, in which each node is the consequence of some rule instance, and its children are the premises of the same rule instance. That node is then an *application* of the rule (schema). Equivalently, all possible proofs are generated inductively by application of rules, where the basis are applications of axioms. In this view, we call proofs also *derivations* of the conclusion from the axioms.

In many type systems [Mit90, CW85, Pie02] a *typing judgement* is a ternary relation

$$\Gamma \vdash e : t$$

between an expression e, a type t and *context* $\Gamma = \{x_i : s_i\}_{i=1}^n$, in which x_i are pairwise distinct identifiers (variables), and s_i are types. (The context Γ is also called the *type assumptions*, *type assignment* or *basis.*) The turnstile $\vdash$ here denotes derivability from assumptions Γ.

This conception takes judgments as atomic objects in deductions, that is there is no intrinsic relation between the left-hand side and right-hand side argument of the turnstile. The typing relation $\vdash$ can thus be regarded as a predicate. This view will be sufficient up to Section 1.2.2, where I will discuss briefly the view of typing judgments as sequents.

1.2.1.4 Type Checking and Type Inference

Strictly speaking, a type *checking* procedure takes as inputs some type assumptions Γ, an expression e and a type t. It then verifies that the relation $\Gamma \vdash e : t$ holds, that is there is a deduction of that judgement using the typing rules. Such a procedure is usually not sensible as a "type checker" of a compiler, as it would require the programmer to supply, besides explicit declarations for parameters, also the result types of every expression in the program.

Type checking in this strict sense is useful, however, when types are read as encodings of logical propositions via the Curry-Howard isomorphism [How80, GTL89, CH88, Bar91, TS00]. Types encode propositions, and terms encode proofs; under this reading type checking implements proof checking. While proof construction (called the *type inhabitation problem* in this context) is usually undecidable, proof checking thus remains decidable and efficient. Furthermore, type checking can be implemented by a program that is small compared to an automated theorem prover. The checker may thus be verified to increase the confidence in the machine-checked proof. This chain of reasoning has found applications in the execution of untrusted code [Nec97, WNKN04].

The other extreme approach consists in type *inference* or (type *reconstruction*), where the programmer does not supply any types in the program, but the compiler computes all necessary type information, including types of function parameters and result types. This mode of "type checking" has been first investigated by Hindley [Hin69] for the λ-calculus and Milner [Mil78] (with Damas [DM82]) for programming languages. It has led to a series of flexible, yet statically type safe languages like ML [Mil78, MTHM97] and Haskell [WB89, Je99].

The type checker of a concrete programming language will usually fall in between these two extremes, as it infers at least the "obvious" types, such as result types of expressions. On the other hand, type inferencers need to check computed types against explicit annotations in the program.

For type inference to be feasible, the type system must be restricted such that every expression can be assigned a single type. (The other possible choice is to refrain from computing explicit types altogether and generate constraints about the possible types only. See Sections 1.2.3.8,1.2.3.6 and 5.2 for a discussion.) When type variables are involved, this aim involves describing the infinitely many possible substitution instances of inferred types with a single type. Again, this is efficient if among these substitution instances, there is one "best" instance, from which all the other instances can be obtained by substitution. This "best" instance is called the *principal type scheme* (or *principal typing*) for the expression [DM82] and type systems for type inference aim at establishing the property that every possible expression has a principal type. This is the case for languages of the ML family by most general unifiers on first-order terms [Rob65]. It continues to hold if constraints are added in a judicious manner ([Jon94, Sul00, Reh97], Section 1.2.3.6).

The relation between type checking and type inference is twofold: In the one direction, given a type inference algorithm and a fully typed program, it is possible to reconstruct the types after erasing them [Mit90]. In the other direction, every type inference algorithm has to return a typing such that a type deduction exists (otherwise it would not be a sound implementation of the given type system). In this sense, the task of type inference is to fill the meta-variables in rule schemas with concrete types, such that a type deduction with valid rule applications is constructed. This view of type inference will be expressed in the design of Tcg.

The latter view can also be imposed on general proof construction with rule schemas, since these are implicitly $\forall$-quantified over their meta-variables.[3] For the example of λ-abstraction, we have

$$\forall s,t.\ \frac{\Gamma, x : s \vdash e : t}{\Gamma \vdash \lambda x.e : s \to t}$$

Before that rule can be applied, the quantifier must be eliminated, which happens by replacing s and t with *unknowns* [Pau94] α, β. (Unknowns are existentially quantified variables. As will be made precise in Section 5.3.2, they closely correspond to type variables

[3]This view abuses notation and anticipates the treatment of rules as objects in Section 1.2. The symbol "$\forall$" is usually reserved for object-level formulae [Pau94, Pfe01].

[Hin69, Mil78, Wan87].) This steps yields a rule with undetermined types:

$$\frac{\Gamma, x : \alpha \vdash e : \beta}{\Gamma \vdash \lambda x.e : \alpha \rightarrow \beta}$$

With this setup, the unknowns will be filled by unification [Pau94, Wan87] as the type check proceeds. Note that this step strictly speaking is possible only if the language of types includes unknowns. Otherwise, one has to argue about the groundness of types in completed derivations. This is the reason why the meta-variables x and e are not shown quantified in the same manner: They are ground and must be determined at the point that the rule is applied.

1.2.2 Comparison with Calculi for Logics

In this section, I briefly sketch natural deduction and sequent calculi as approaches to defining logical calculi.[4] The intention of this presentation is to establish terminology for the later discussion of the Tcg design. I demonstrate that despite superficial correspondences, conventional typing rules fall between the two approaches when considering the details of the formalisms. Hence, a direct application of established formalisms to Tcg is not possible. However, brief comparisons with specific typing rules already in this section motivate that the design of Tcg benefits from proof-theoretical considerations for logics. The presentation is based on [Gen35, Pra65, NP01, TS00, GTL89].

The formulations of many type systems are based on a judgement written $\Gamma \vdash e : t$, which assigns type t to expression e in context Γ (Section 1.2.1.3), and they use deduction rules to specify the typing judgement. The precise structure of the resulting proofs does not seem to have been investigated widely in the type systems literature. The main goal there is to prove soundness of the deduction system relative to some semantics of programs. (For examples in varying settings, see [Mil78, Tof90, Mit91, WF92].) The formalization of the typing relation appears as a meta-level vehicle towards that end. One exception to this tendency is Mitchell and Plotkin's treatment of abstract data types by analogy to intuitionistic existential quantification [MP88]. In Section 4.3 of that article, the authors motivate their choice of typing rules by comparison with the logical rules.

Through the Curry-Howard-isomorphism [How80], the proof structure of typing judgments has a well-studied counterpart in logic [ML84, CH88, GTL89, Bar91, BG00, Ghi99, Tro99, TS00]: The simply typed λ-calculus is isomorphic to natural deduction systems (with labeled assumptions), where β-reduction corresponds to cut-elimination. These studies are mostly concerned with the correspondence of cut-elimination and β-reduction, not with natural formulations of type systems, and they do not consider type checking of programming languages.

[4] I omit axiomatic (or Hilbert-style) systems because I wish to study the structure of inference rules. Note, however, that in the context of logical frameworks (Section 5.3, also [Pau86, Pau89]) the object logic is encoded as axioms in the meta logic, which provides a fixed set of inference rules.

1.2.2.1 Natural Deduction

Natural deduction, introduced by Gentzen [Gen35], intends to model and formalize the proof methods found in mathematical texts. In this system, derivation trees are labeled with formulae. (But see Remark 1.2.1.)

Propositional Fragment Each logical connective is characterized by a pair of *introduction* and *elimination rule*. The introduction rule describes the connective in terms of the proof(s) that must be constructed to justify the connective in a formula. The elimination rule describes the use that can be made of a proven formula with that connective. For instance, the conjunction $\wedge$ has rules

$$\frac{A \quad B}{A \wedge B}(\wedge\mathrm{I}) \qquad \frac{A \wedge B}{A}(\wedge\mathrm{E}_1) \qquad \frac{A \wedge B}{B}(\wedge\mathrm{E}_2)$$

The second characteristic of natural deduction is its treatment of *assumptions* (or *hypotheses*, hence *hypothetical reasoning*). Assumptions are noted above the derivation that may use them; they then appear as leaves of the derivation. For instance, the connective $\supset$ for implication is introduced as:

$$\frac{\begin{array}{c}[A]\\ \vdots \\ B\end{array}}{A \supset B}\,(\supset\mathrm{I})$$

The assumption A from this subproof is *discharged* at the ($\supset$I) application, meaning that it is not considered an assumption further down in the derivation. Assumption that are not discharged within the derivation are called *open*. (For an inductive definition of open and closed assumptions, see [NP01, Definition 8.1.1], [TS00, Definition 2.1.1]). In the graphical notation, the distinction is made clear by surrounding discharged assumptions with brackets. Discharges can also concern unused assumptions, by introducing $D \supset A$, where the derivation of A does not depend on D at all. Such discharges are called *vacuous*.

To make the point of discharge explicit, assumptions can be labeled (with names or numbers), and the label of the discharged assumption is noted at the application of the rule. The explicit annotation with discharged assumptions is crucial to make the deductions precise [TS00, Section 1.3]. For instance, there are two derivations of $A \supset A \supset A$ (where all applied rules ($\supset$I) are indicated by the discharged assumption; the underlined discharges are vacuous).

$$\dfrac{\dfrac{\dfrac{\dfrac{A^u}{A \supset A}\,\underline{v} \qquad A^w}{A}}{A \supset A}\,u}{A \supset (A \supset A)}\,w \qquad \dfrac{\dfrac{\dfrac{\dfrac{A^u}{A \supset A}\,u \qquad A^v}{A}}{A \supset A}\,v}{A \supset (A \supset A)}\,\underline{w} \tag{1.2.1}$$

The elimination rule of $\supset$ is modus ponens:

$$\frac{A \supset B \quad A}{B}\,(\supset\mathrm{E})$$

Inversion Principle Negri and von Plato [NP01, Section 1.2] show how the elimination rule(s) for a connective can be obtained from its introduction rule via a general *inversion principle*: "Whatever follows from the direct grounds for deriving a proposition must follow from that proposition." [NP01, Page 6] (for earlier formulations see also [Pra65, Section II.1], [Gen35, Section II.5.13][5]). In short, this principle ensures that by introducing a connective, no information is lost compared to using the already derived premises of the introduction rule directly (e.g. [Pra65, *inversion theorem*, Section II.1]). For example, in the case of $\wedge$, the elimination rule thus found is

$$\frac{A \wedge B \qquad \begin{array}{c}[A, B]\\ \vdots\\ C\end{array}}{C}\,(\wedge\mathrm{E}')$$

According to rule ($\wedge$I), the direct grounds for introducing $A \wedge B$ are precisely A and B, which by this rule may be used to infer C. Note that ($\wedge$E$'$) is a generalization of ($\wedge$E) in that it allows to derive all consequences of A, B, not only A and B themselves. (The rule is, however, not stronger than ($\wedge$E), since the same judgement C under assumptions A, B can be obtained by applying ($\supset$I) twice on $A \supset B \supset C$. Using ($\supset$E) twice with ($\wedge$E) then yields the conclusion C.)

Natural Deduction in Sequent Style The formalization of open assumptions in natural deduction is not very concise, as the open assumptions for a judgement can be determined only by examining the entire deduction tree above it. A solution is to keep track of open assumptions in each judgement, which leads to *Natural Deduction in Sequent Style* [TS00, Section 2.1.8][NP01, Sections 1.3, 5.2, Chapter 8][GTL89, Sections 5.3, 5.4]. Here, derivability of a formula is captured by $\Gamma \Rightarrow A$, where Γ is a multi-set of the open, labeled assumptions in the derivation of A. It is important to note that this definition does *not* include unused assumptions in Γ (see derivation (1.2.1) above). As Negri and von Plato [NP01, Chapter 8] point out, this is the main difference to the *meta-level* notion of derivability under assumptions $\Gamma \vdash A$, which denotes the existence of a (natural deduction) derivation of A from assumptions *contained in* Γ.

The inference rules of natural deduction in sequent style are derived from those of natural deduction. They deviate sufficiently from the classical sequent calculi (Section 1.2.2.2) to afford a forward reference at this point: First, there are no left-rules, but elimination rules for the connectives. Second, according to the definition of Γ detailed above, the antecedent of a rule's conclusion is the concatenation of the antecedents of its premises.

[5]Gentzen already conjectures that "[b]y making these ideas more precise, it should be possible to display the E-inferences as unique functions of their corresponding *I*-inferences, on the basis of certain requirements." [Gen35, Section II.5.13]

(This feature is termed “sequent calculi with independent contexts” by Negri and von Plato [NP01, Section 5.1].) In the following example rules [TS00, Section 2.1.8], the subscript S indicates “sequent style”; the notation $[u : A]$ means that assumption $u : A$ may be present or absent (if it is not used in the derivation) in the context, and u does not occur in Γ.

$$\frac{\Gamma[u : A] \Rightarrow B}{\Gamma \Rightarrow A \supset B}(\supset\mathrm{I}_S) \qquad \frac{\Gamma \Rightarrow A \supset B \quad \Delta \Rightarrow A}{\Gamma\Delta \Rightarrow B}(\supset\mathrm{E}_S)$$

$$\frac{\Gamma \Rightarrow A \quad \Delta \Rightarrow B}{\Gamma\Delta \Rightarrow A \wedge B}(\wedge\mathrm{I}_S) \qquad \frac{\Gamma \Rightarrow A_0 \wedge A_1}{\Gamma \Rightarrow A_i}(\wedge\mathrm{E}^i_S)$$

Comparison with Type Systems Type systems are often specified in two equivalent formulations (e.g. [DM82, Jon94, Sul00]): The *logical* formulation describes how type constructors can be *introduced* and *eliminated*, thus following the spirit of natural deduction. The second form is *syntax directed*, meaning that there is exactly one derivation rule for each programming language construct. The equivalence is obvious for the simply typed λ-calculus, where function application corresponds to elimination of the function type constructor, and λ-abstraction corresponds to introduction of that constructor. It breaks down, for example, for ML-style `let`-polymorphism [Mil78]: Because polymorphism is implicit, there is no syntactically marked point for introducing and eliminating the $\forall$ quantifier. The solution is to adopt a proof normal form, in which $\forall$ introduction occurs only at `let`-bindings and $\forall$ elimination is executed at variable references. (See also [CW85, Pfe88, CH88, Car93, OL96] for the relation to explicit polymorphism with Λ-abstraction over types.)

Type systems usually do not make use of hypothetical reasoning explicitly. The open assumptions concern program variables, and they are captured in context Γ of typing judgments. However, these judgments $\Gamma \vdash e : t$ bear only a superficial correspondence with natural deduction in sequent style: The context Γ contains the assumptions on the types of *all* variables bound in the surrounding term, not just the used ones. This difference brings the typing judgement nearer to the meta-level reading of derivability under assumptions (see [NP01, Chapter 8]). The difference can be closed by a technique used by Mitchell and Plotkin [MP88, Section 4.3]: From the context Γ of the typing judgement, select only those entries $x : s$, for which x appears free in the expression e to be typed.

Concluding, natural deduction is not the only ingredient to the deduction structure of type systems: Although superficially $e : t$ can be seen as a proposition, and Γ contains assumptions of the same form, the correspondence breaks down when considering the details and intentions of the different formalisms.

Quantifiers Quantifiers in natural deduction are introduced and eliminated like the connectives. We start with the $\forall$ quantifier.

$$\frac{A[y/x]}{\forall x A}(\forall\mathrm{I}) \qquad \frac{\forall x A}{A[t/x]}(\forall\mathrm{E})$$

The most important point of these rules is in the side-condition on the choice of the *eigenvariable* (or *proper variable*) y in rule ($\forall$I) [NP01, Section 4.1.(b)]: Either $x \equiv y$, or

y must not appear free neither in A nor in the open assumptions from which $A[y/x]$ has been derived. This restriction ensures that y stands for an arbitrary individual.

In the intuitionistic interpretation of existential quantification (e.g. [ML84]), the introduction of $\exists$ must be justified by exhibiting an example term t, for which the desired quantified proposition can be derived. In elimination, the desired conclusion from an existentially quantified proposition must be derived for an arbitrary y, representing that unknown example term [NP01, Section 4.1.(b)]:

$$\frac{A[t/x]}{\exists x A}(\exists \mathrm{I}) \qquad \frac{\exists x A \qquad \begin{array}{c}[A[y/x]]\\ \vdots \\ C\end{array}}{C}(\exists \mathrm{E})$$

The restriction on y in ($\exists$E) is that it does not occur free in $\exists x A$, C, nor in any (open) assumption that C depends on [NP01, Section 4.1.(b)]. Again, this restriction entails that no further assumptions on the individual y can be made.

Comparison with type systems The quantifier rules for $\forall$ parallel the type theoretic treatment of polymorphism. Introduction and elimination correspond to the programming language notions of *generalization* and *instantiation.*

$$\frac{\Gamma \vdash e : s \quad \alpha \notin \mathbf{FV}(\Gamma)}{\Gamma \vdash \forall \alpha . s}(\mathrm{GEN}) \qquad \frac{\Gamma \vdash \forall \alpha . s}{\Gamma \vdash e : s[t/\alpha]}(\mathrm{INST}) \tag{1.2.2}$$

The eigenvariable condition also parallels the logical formulation in sequent style natural deduction, where Γ corresponds to the open assumptions of a derivation. However, the correspondence is not precise, since a direct translation would yield a technically weaker proviso[6]

$$\alpha \notin \mathbf{FV}(\Gamma|_{\mathbf{FV}(e)}) \tag{1.2.3}$$

The introduction and elimination of $\forall$ are rarely used in programming languages, since only some experimental languages with explicit polymorphism (e.g. [Car93, CH88, Bar91]) have syntactically marked points for their application. As mentioned above, the logical formulation of a type system is usually substituted with an equivalent syntax directed version. The ambiguity about applications of these rules is overcome by using normal forms of derivations: (GEN) is allowed only in connection with `let` bindings, where it generalizes all type variables not excluded by the side-condition $\alpha \notin \mathbf{FV}(\Gamma)$. Rule (INST) is applied only at variable references, where it eliminates all existing quantifiers.

Mitchell and Plotkin [MP88] introduce the $\exists$ quantifier as a type-theoretic formulation of data abstraction. In Section 2, they motivate their interpretation by the desired effect of data abstraction, referring to existing languages for justification. In Section 4, they refer to the formulae-as-types notion for a second justification of their typing rules from the intuitionistic interpretation. In Section 4.3, they treat existential quantification from this direction, by briefly comparing their typing rules to those from [Pra65].

[6]For the practical purpose of type inference, the proviso is not weaker: Type variables from Γ can only appear in the type s if e contains a variable from Γ.

Mitchell and Plotkin's introduction rule for $\exists$ is tied to the syntactic construct **pack** [MP88, Section 3.4], which is comparable to module definitions in programming languages. In the following rule, t is the representation type of the abstract type α.[7]

$$\frac{\Gamma \vdash M : s[t/\alpha]}{\Gamma \vdash \mathbf{pack}\ t\ M\ \mathbf{to}\ \exists\alpha.s : \exists\alpha.s}$$

Note how the example type t parallels the example term t in ($\exists$I). The (typing-) justification for the substitution $[t/\alpha]$ has already been given by Morris [Mor73]: Within the definition of the abstract type, the abstract name α is convertible to and from the representation type t.

Abstract data types are introduced with a limited scope in which the new type operation names may be used [MP88, Section 3.4].[8]

$$\frac{\Gamma \vdash M : \exists\alpha.s \quad \Gamma, [x : s] \vdash N : t}{\Gamma \vdash \mathbf{abstype}\ \alpha\ \mathbf{with}\ x : \sigma\ \mathbf{is}\ M\ \mathbf{in}\ N : t}$$

This rule comes with the following variable restriction [MP88, Section 3.4] (transliterated according to Section 1.2.1.1): "provided that α is not free in t or the type $\Gamma(y)$ of any free $y \neq x$ occurring in N." This proviso is the same as the one for ($\exists$E) (see also (1.2.3)): The type t corresponds to proposition C, the open assumptions above C are precisely those variables from Γ that occur free in N. The first proviso $y \notin \mathbf{FV}(\exists x A)$ of ($\exists$E) is not necessary, as (after renaming) α is the bound name of the existential type.[9]

1.2.2.2 Sequent Calculi

Sequent calculi, introduced by Gentzen [Gen35], aim at formalizing reasoning with hypotheses [Gen35, Section III.1.1]. Their assertions take the form of *sequents*

$$A_1 \mathinner{..} A_n \Rightarrow B_1 \mathinner{..} B_m$$

where A_i, B_j are formulae, possibly $n = 0$ and/or $m = 0$. Derivations are then labeled with sequents. (But see Remark 1.2.1.) $A_1 \mathinner{..} A_n$ is the *antecedent* of the sequent, $B_1 \mathinner{..} B_m$ its *succedent*. Unlike the turnstile $\vdash$, the arrow $\Rightarrow$ is an object-level symbol, derivations are concerned with assertions $\vdash \Gamma \Rightarrow \Delta$.

The classical reading of sequents is denotational [Gal86]: A sequent is interpreted as a formula $\bigwedge_{i=1}^{n} A_i \supset \bigvee_{j=1}^{m} B_j$. Negri and von Plato [NP01, Page 47] point out that the A_i may also be read as *open assumptions* while the B_j can be read as *open cases*.

[7]Unlike in most programming languages, however, the name of the abstract type can be changed by α-conversion once the module definition is complete.

[8]The presentation corrects a typing mistake in the rule (Section 3.4, page 483): The original has in the conclusion "**abstype** s **with**" but uses "$\exists t.\sigma$" in the premise, contradicting the condition (AB.1), Section 2, page 474, which is explicitly referenced as a justification for the typing rule.

[9]This forced renaming can be avoided by representing identifiers as names with *stamps* (e.g. [Ler95])): During α-conversion, only the stamp part is changed, the name part remains the same for access of the data type's components.

The derivations in sequent calculus (e.g. [NP01, Section 2.2, system **G3ip**]) start, at the leaves, with *initial sequents* (or *axioms*) of the form

$$A, \Gamma \Rightarrow A$$

The sequent at the derivation's root is called the *endsequent.*

In what follows, we will be concerned with the single-succedent sequents, which capture intuitionistic derivability of B under assumptions $A_1 \mathinner{..} A_n$:

$$A_1 \mathinner{..} A_n \Rightarrow B$$

Like natural deduction in sequent style, they make explicit the assumptions usable in the derivation of B. However, at rules with more than one premise, the contexts of the premises are *shared.* Hence, $A_1 \mathinner{..} A_n$ are not the open assumptions of the derivation of B, but the assumptions *potentially* usable in that derivation (cf. Section 1.2.2.1).

The introduction rules of natural deduction become *right rules.* They capture the proof obligations for the logical connectives. For instance, to prove $A \wedge B$, one has to prove both A and B.

$$\frac{\Gamma \Rightarrow A \qquad \Gamma \Rightarrow B}{\Gamma \Rightarrow A \wedge B}(\wedge\mathrm{R})$$

The elimination rules are replaced [GTL89, Sections 5.3, 5.4] by *left rules*, which describe the handling of a connective in the antecedent of a sequent.

$$\frac{A, B, \Gamma \Rightarrow C}{\Gamma, A \wedge B \Rightarrow C}$$

Three observations about left rules are relevant to the later treatment of type systems.

1. The connective $\wedge$ is explained in terms of two extra-logical constructs: The two premises A and B in ($\wedge$R) express the required construction of proofs for both A *and* B. (This *and* is a meta-level construct.) The rule ($\wedge$L) explains $\wedge$ in terms of the comma (multi-set union) in the antecedent.
2. The left rules can be read as *forward-reasoning* within the antecedent of a sequent: To prove C under $A \wedge B$, we can assume both A and B. This corresponds to the natural deduction formulation

$$\cfrac{\begin{array}{c}\cfrac{[A \wedge B]}{A} \qquad \cfrac{[A \wedge B]}{B} \\ \vdots \\ C\end{array}}{A \wedge B \supset C}$$

   The two derivations at the leaves transform the assumption $A \wedge B$ into the more basic forms A and B, which may then perhaps be more easily reached by *backwards* proofs from C.

3. Unlike elimination rules, left rules do not introduce a new formula in the premises that was not present in the consequence. Sequent calculi thus have the *subformula property* [NP01, p. 15]: "All formul[ae] in a sequent calculus derivation are subformul[ae] of the endsequent of the derivation." This property is very desirable for proof search, because no arbitrary choices of terms or formulae need to be made within the derivation.

An interesting case is the connective $\supset$ of implication. Its meaning is explained in terms of the extra-logical, object-level symbol $\Rightarrow$:

$$\frac{\Gamma, A \Rightarrow B}{\Gamma \Rightarrow A \supset B}(\supset\mathrm{R}) \qquad \frac{A \supset B, \Gamma \Rightarrow A \qquad B, \Gamma \Rightarrow C}{A \supset B, \Gamma \Rightarrow C}(\supset\mathrm{L})$$

The rule ($\supset$R) introduces $\supset$ by adding the premise A to the antecedent (set of assumptions) for a proof of B. Reading this rule downward, the assumption A can be removed from the set of assumptions (i.e. in natural deduction terminology, it can be discharged) by writing it as the premise of $\supset$.

The rule ($\supset$L), the counterpart of natural deduction cut (or modus ponens), employs the implication $A \supset B$ by first proving A, and then deriving C from B. Note that also ($\supset$L) preserves the subformula property. This is contrary to rule ($\supset$E) in natural deduction, where the cut formula cannot be obtained by inspection of the conclusion. Again, sequent calculi seem more appropriate for automatic provers, as the search space is limited.

Besides the left rules, which allow modifications of the antecedent, sequent calculi have a second characteristic: Antecedent and succedent of a sequent are perceived as sequences or multisets, rather than sets, of formulae. (We deviate from the restriction to single-succedent, intuitionistic sequent calculi to illustrate the symmetry.) Then *structural* properties of proof systems can be made explicit in proofs. The basic *structural rules* [Gen35, Section III.1.21] are *thinning*, *contraction* and *interchange*, each of which can be applied to both antecedent and succedent.

$$\begin{array}{lcc}
\text{thinning} & \dfrac{\Gamma \Rightarrow \Delta}{A, \Gamma \Rightarrow \Delta} & \dfrac{\Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta, A} \\
\text{contraction} & \dfrac{A, A, \Gamma \Rightarrow \Delta}{A, \Gamma \Rightarrow \Delta} & \dfrac{\Gamma \Rightarrow \Delta, A, A}{\Gamma \Rightarrow \Delta, A} \\
\text{interchange} & \dfrac{\Delta, A, B, \Gamma \Rightarrow \Phi}{\Delta, B, A, \Gamma \Rightarrow \Phi} & \dfrac{\Gamma \Rightarrow \Delta, A, B, \Phi}{\Gamma \Rightarrow \Delta, B, A, \Phi}
\end{array}$$

The rules explicate the treatment of the sequences in a sequent as *sets* of assumptions and formulae to be proven. Other perceptions of sequents are possible; restrictions in the set of structural rules then lead to substructural logics.

Quantifiers The quantifier rules of natural deduction introduced an eigenvariable condition that refers to the open assumptions of a judgement. With the explicit statement of the open assumptions in sequents, their formulation thus becomes [NP01, Section 4.1.(c)]

$$\frac{A(t/x), \forall xA, \Gamma \Rightarrow C}{\forall xA, \Gamma \Rightarrow C}(\forall \mathrm{R}) \quad \frac{\Gamma \Rightarrow A(y/x)}{\Gamma \Rightarrow \forall xA}(\forall \mathrm{R})$$

$$\frac{A(y/x), \exists xA, \Gamma \Rightarrow C}{\exists xA, \Gamma \Rightarrow C}(\exists \mathrm{R}) \quad \frac{\Gamma \Rightarrow A(t/x)}{\Gamma \Rightarrow \exists xA}(\exists \mathrm{R})$$

where in (∀R), variable y must not occur free in Γ, $\forall xA$, and in (∃L), y must not occur free in $\exists xA$, Γ and C. In other words, in these rules the eigenvariable must not occur in the lower sequent.

Comparison with Type systems Type systems, as found in the literature, do not usually contain left rules. However, Mitchell [Mit90, Section 2.2.1] points out that the notation $\Gamma \vdash e : s$ is often read as a sequent with antecedent Γ, although its common interpretation is natural deduction derivability under assumptions Γ. (See the comparison in Section 1.2.2.1)

Consequently, the forward reasoning enabled by left rules must be made explicit in typing rules. For instance, the logical system (1.2.2) for ML-style implicit polymorphism needs the following rule for variable references, which combines the "lookup" of the variable x in context Γ with an instantiation of the bound type variables:

$$\frac{x : \forall \tilde{\alpha}.t \in \Gamma}{\Gamma \vdash x : t[\tilde{s}/\tilde{\alpha}]}$$

This can be expressed by forward reasoning, or left rules ((INST)* denotes multiple, possibly zero applications of rule (INST)):

$$\cfrac{\cfrac{x : \forall \tilde{\alpha}.t \in \Gamma}{\Gamma \vdash x : \forall \tilde{\alpha}.t}(\mathrm{VAR})}{\Gamma \vdash x : t[\tilde{s}/\tilde{\alpha}]}(\mathrm{INST})^*$$

Despite the possibility of such reductions of typing rules to more basic constructs, left rules come at the severe price of being inadequate for backwards reasoning, which can be seen as the main mode of proof construction in syntax-directed type checkers [Car87]. Therefore, for the present work, I will not be concerned with left rules, but introduce explicit forward reasoning steps where necessary.

The discharge of assumptions in the ⊃R has also been noted in constraint-based type systems as "shifting of assumptions from the left-hand side to the right-hand side of the turnstile" (cf. [Jon94, Sul00], Sections 1.2.3.6, 5.2.1).

Substructural restrictions in the area of type systems model restrictions on the use of values, leading for instance to linear types (following [Gir87]), which enforce that every variable in a context can be referenced only once, which facilitates memory management [Wad90]. In the subsequent text, we will be concerned with *sets of type assumptions* only, since this conception prevails in most type systems.

1.2.3 Proofs

This section presents TCG's notion of *proof*, which will be defined in Chapter 2, by way of motivation. Each subsection presents a specific feature found in type systems in the form of a prototypical typing rule. Then, it shows how this construct is incorporated to the design of TCG as a general principle of proof construction. Finally, the principle is applied to similar constructs from typing to indicate its usefulness apart from the original motivation.

The overall design goals in this process are conceptual simplicity and a minimal set of features. Where a new feature can be reduced to one already introduced, it is done. In some instances, this comes at the price that the original typing rule cannot be written down directly in TCG. However, I will argue that in these cases TCG's formulation clarifies the intention of the original rule and establishes a connection with the logical calculi in Section 1.2.2.

1.2.3.1 Typing Rules

Typing rules are at the object-level in TCG. They will in general have premises $P_1 \mathinner{..} P_n$ and a conclusion C. The meta-variables contained in the typing rules (Section 1.2.1.3) are noted as bound variables in TCG rules. We say that the rule is *quantified over* these variables. TCG rules thus have the form

$$\forall v_1 \mathinner{..} v_m \big[P_1 \mathinner{..} P_n\big] \Longrightarrow C$$

which we also render as

$$\forall\big(v_1 \mathinner{..} v_m\big)\ \frac{P_1 \mathinner{..} P_n}{C}$$

1.2.3.2 Static Scope

Statically typed programming languages also have static scope: The accessible identifiers at each program point must have been introduced in some surrounding construct. It is obvious that the typing context Γ (Section 1.2.1.3) serves to capture exactly the scoping behaviour and available identifiers (if the structure of the typing derivation is aligned with the nested scopes of the program, of course). The prototypical rule is variable lookup.

$$\frac{x : t \in \Gamma}{\Gamma \vdash e : t}$$

If we were to introduce the predicate $x : t \in \Gamma$ as a primitive notion in TCG, we would fix the form of typings assignable to identifiers, excluding for example type schemes with constraints $\forall\tilde{\alpha}.\pi \Rightarrow t$ [Jon94].

Instead, we will make the context Γ itself hold TCG rules, and we make rule application from Γ the primitive operation. Since in general a rule has premises, we have an application of the form

$$\frac{\Gamma_1 \vdash P_1[\tilde{t}/\tilde{v}] \mathinner{..} \Gamma_n \vdash P_n[\tilde{t}/\tilde{v}]}{\Gamma \vdash C[\tilde{t}/\tilde{v}]} \quad \text{where} \quad \begin{array}{c}\left(\forall\tilde{v}\ \dfrac{P_1 \mathinner{..} P_n}{C}\right) \in \Gamma \\ t_1 \mathinner{..} t_n \text{ terms}\end{array} \tag{1.2.4}$$

We leave open for the moment how the Γ_i are derived. A type assumption $x : t$ is encoded as an axiom, that is a rule without premises:[10]

$$\frac{}{x : t}$$

First-Order Polymorphic Values First-order polymorphic values can obviously be encoded by quantified rules. For instance, the operation `fst`, which extracts the first component of a pair of values, has the type expressed by

$$\forall \alpha, \beta \ \frac{}{\texttt{fst} : \langle \alpha, \beta \rangle \to \alpha}$$

Type Names In the same manner, we can capture a type name c that is defined in the current environment by assigning it kind $*$.

$$\frac{}{c :: *}$$

Type constructors with parameters, for instance lists and pairs, have higher kinds of the form $\kappa \to \kappa'$. The well-formedness of type expressions is then checked by recursively checking all applications of type constructors.

Type Abbreviations A declaration

$$c(\alpha_1 \mathinner{..} \alpha_n) = s[\alpha_1 \mathinner{..} \alpha_n]$$

defines the type constructor c as an abbreviation that is to be expanded into s, replacing the identifiers α_i with the actual parameters. This definition is captured by a rule[11]

$$\forall (v_1 \mathinner{..} v_n) \frac{}{c(v_1 \mathinner{..} v_n) \equiv s[v_1 \mathinner{..} v_n / \alpha_1 \mathinner{..} \alpha_n]}$$

Return Types If Γ contains a special rule $\overline{\mathrm{ret}(r)}$ capturing the return type of the current function, then the imperative `return` statement is modeled by a rule:

$$\frac{\Gamma \vdash e : r \qquad \Gamma \vdash \mathrm{ret}(r)}{\Gamma \vdash \textbf{return } e}$$

1.2.3.3 Application

Probably the most fundamental language construct is function application. The typing rule must check that the applied object is a function and that the type of the argument

[10] We will usually drop the line delimiting the empty premises from the conclusion.

[11] $s[v_1 \mathinner{..} v_n/\alpha_1 \mathinner{..} \alpha_n]$ is the meta-level simultaneous substitution of $v_1 \mathinner{..} v_n$ for $\alpha_1 \mathinner{..} \alpha_n$ throughout s.

matches the type expected by the function. The result of the application is the result that the function is specified to return:

$$\frac{\Gamma \vdash f : s \to t \quad \Gamma \vdash e : s}{\Gamma \vdash (f\ e) : t}\,(\text{apply})$$

This rule partially answers the questions about how the Γ_i of the premises in (1.2.4) are obtained: If nothing else is specified, they are the same as the context Γ of the judgement where the rule is applied. A second characteristic, as explicated by Wand [Wan87], is the implied equality constraint between the formal and the actual argument.

Variations Variations of the application for multiple arguments are a primitive for multiple arguments, as in imperative languages, and tuple arguments and curried application as in Standard ML.[12]

$$\frac{f : s_1 \mathinner{\ldotp\ldotp} s_n \to^n t \quad e_1 : s_1 \mathinner{\ldotp\ldotp} e_n : s_n}{(f\ e_1 \mathinner{\ldotp\ldotp} e_n) : t}\,(\text{apply}_n)$$

$$\frac{f : (s_1 \mathinner{\ldotp\ldotp} s_n) \to t \quad (e_1 \mathinner{\ldotp\ldotp} e_n) : (s_1 \mathinner{\ldotp\ldotp} s_n)}{(f\ (e_1 \mathinner{\ldotp\ldotp} e_n)) : t}\,(\text{apply}_{\text{tuple}})$$

$$\frac{f : s_1 \to \cdots \to s_n \to t \quad e_1 : s_1 \mathinner{\ldotp\ldotp} e_n : s_n}{((f\ e_1) \ldots e_n) : t}\,(\text{apply}_{\text{curry}})$$

Procedures and Statements Imperative procedure calls fit the function application pattern by introducing type `void` that does not have any values [KR88]. In functional languages with imperative features, those functions which are called for their side-effects only receive return type **unit**, which differs from **void** in that it has a single value `()` (hence the type name).

Primitive Operations Primitive operations are obviously subsumed by considering their specialized typing rules as operator definitions:

$$\frac{i : \texttt{Int} \quad j : \texttt{int}}{i + j : \texttt{int}} \quad \text{becomes} \quad \frac{}{+ : \texttt{Int} \to \texttt{Int} \to \texttt{Int}}$$

Ad-hoc Overloading Operator overloading requires at least the enumeration of all possible typings and can be reduced to a search for possible proofs [GR80, Cor82, Bak82]. The selection of a "best" match [Str97], however, is problematic. (See Section 4.3.4.)

Method invocation The type-theoretical treatment of object-orient languages, at least since [Red88], has focused on encoding their specific features into domains, whose typing properties are understood [Red88, CHC89, PT94, EST95a, EST95b, AC96, Cas95, Pie02].

[12]An n-ary function $\lambda x_1 \mathinner{\ldotp\ldotp} x_n.e$ is translated to an n-fold abstraction $\lambda x_1 \mathinner{\ldotp\ldotp} \lambda x_n.e$ [Thi94].

Method invocations in conventional class-based [AC96, GJS00, Str97] object-oriented languages differ from ordinary function application in the special role of the first `this` (or `self`) argument. In theoretical work, it is modeled as a fixpoint construction [CHC89, Section 2]:

$$C = \lambda \mathit{self}.\{m_1 = e_1 \mathinner{..} m_n = e_n\} \tag{1.2.5}$$

$$I = \mathbf{Y}(C) \tag{1.2.6}$$

$$C' = \lambda \mathit{self}.S(\mathit{self})\ \mathbf{with}\ \{m'_1 = e'_1 \mathinner{..} m'_m = e'_m\} \tag{1.2.7}$$

The term C in (1.2.5) models a class, which describes the structure of its instance objects as records. In (1.2.6), an instance I is created by taking the fixpoint of its class C. This step binds the name *self*, which may occur in the methods $e_1 \mathinner{..} e_n$, to reference the object's record. Finally, a derived class C' is created from some superclass in (1.2.7) by extending and modifying the record of fields and methods. The typing rule for method calls is derived from the typing rules for application and record selection.

The above procedure is contrary to a compiler-construction view of objects. Here, the method call is indicated by a special syntax:

$$e.m(e_1 \mathinner{..} e_n)$$

The compiler (e.g. [gcj04, `typeck.c`]) searches the class tree explicitly for a method with matching name m and signature for types of $e_1 \mathinner{..} e_n$. A corresponding typing rule is (e.g. [ON99], [AC96, Section 7.2, Section 12.4]):

$$\frac{\begin{array}{l}\Gamma \vdash e : \mathtt{class}(C) \\ \Gamma \vdash C :: \mathtt{methods}(m_1 : s_{11} \mathinner{..} s_{1k_1} \to t_1 \mathinner{..} m_n : s_{n1} \mathinner{..} s_{nk_n} \to t_n) \\ m = m_i \\ e_1 : s_{i1} \mathinner{..} e_n : s_{in}\end{array}}{e.m(e_1 \mathinner{..} e_n) : t_i}\,\text{(method-call)} \tag{1.2.8}$$

An indirection through the class name C is required in the first premise [ON99]: The method signatures may refer to class C again, or they may be even mutually recursive with other classes, so the $\mathtt{methods}(\ldots)$ would need to have a recursive structure, if we were to attach it to e directly [CHC89, CHC89].

Type Expressions The application of type constructors can be checked for well-formedness by introducing kinds, which can be thought of as "types of types" [Bar91]. The rule for application of type constructors is (following [Bar91, Definition 3.2]):

$$\frac{\Gamma \vdash F :: \kappa \to \kappa' \quad \Gamma \vdash T :: \kappa}{\Gamma \vdash F(T) :: \kappa'}$$

Jones [Jon95] uses a kind layer to ensure that the constructor variables of his Haskell extension always lead to well-formed instantiations.

1.2.3.4 Functions

Type checking a function $\lambda x : s.e$ requires a type check of the body e with access to the parameter x of type s. Within e, the identifier x is bound to the function's parameter, the identifiers x bound at outer scopes become invisible.

$$\frac{\Gamma_x, x : s \vdash e : t}{\Gamma \vdash \lambda x : s.e : s \to t} \tag{1.2.9}$$

The removal of the previous entries for x in Γ can be avoided by renaming the bound variable x before applying the rule. However, this requires a separate binding analysis on the parse tree and thus contradicts our intention of processing raw parse trees (Section 1.1).

The main novelty of the abstraction rule is that the context in the premise is a modified version of the context in the conclusion. With the encoding of type assumptions as rules (Section 1.2.3.2), the operation Γ_x requires the removal of all rules which have the form

$$\forall \tilde{v} \, \frac{}{x : s'}$$

Subsequently a new rule for $x : s$ has to be introduced. The notion of *context modifier* comprises these desired modifications. Let M_i be context modifiers. Then the general form of a rule is

$$\forall \tilde{v} \, \frac{M_1 \vdash P_1 \mathinner{.\,.} M_n \vdash P_n}{C}$$

A context modifier M will hold instructions $+R$ to add some rule R and $-s$ to remove all rules matching a selector s (Section 3.1.1). Writing the application of M to context Γ as $\Gamma@M$, we have a new form of rule application, which adds to (1.2.4) the specification of the contexts in premises.

$$\frac{\Gamma@(M_1\sigma) \vdash P_1\sigma \mathinner{.\,.} \Gamma@(M_n\sigma) \vdash P_n\sigma}{\Gamma \vdash C\sigma} \qquad \begin{array}{l} \left(\forall \tilde{v} \, \dfrac{M_1 \vdash P_1 \mathinner{.\,.} M_n \vdash P_n}{C}\right) \in \Gamma \\ t_1 \mathinner{.\,.} t_n \text{ terms,} \\ \sigma := [\tilde{t}/\tilde{v}] \end{array} \tag{1.2.10}$$

Multiple Arguments As in Section 1.2.3.3, functions and procedures with multiple arguments can be accommodated by n-ary functions, functions that take tuples as arguments or currying. The treatment of the return type in Pascal-like functions has already been given in Section 1.2.3.2.

Methods Methods of classes have access to a much richer context than single functions or procedures, as they may refer to instance variables, other methods and global variables. Much of the new complexity is due to the recursive references, which are resolved in a two-pass process (Section 4.4). *Static methods* relate to functions and application directly (albeit in a context extended by the static data members of the class) [Lip96].

Type Abstraction The abstraction over values can be paralled by an abstraction $\Lambda t.M$ over types [CH88, Bar91]. The kinds check for type abstraction then parallels the type check for ordinary abstraction.[13]

Definitions Deferring polymorphism to Section 1.2.3.5, a definition **let** $x = e$ **in** e' requires the same modifications of contexts as function application. Mutually recursive definitions like **letrec** allow the defining expressions to reference defined names. They are also contained in the current design. Here is a prototypical rule

$$\frac{\begin{array}{l}\text{for } i = 1 \mathinner{..} n : \Gamma_{x_1 \mathinner{..} x_n} \cup \{x_1 : s_1 \mathinner{..} x_n : s_n\} \vdash e_i : s_i \\ \Gamma_{x_1 \mathinner{..} x_n} \cup \{x_1 : s_1 \mathinner{..} x_n : s_n\} \vdash e' : t\end{array}}{\Gamma \vdash \textbf{letrec } x_1 = e_1 \mathinner{..} x_n = e_n \textbf{ in } e' : t}$$

The same approach also yields recursively defined types, for instance algebraic data types in ML [MTHM97] or mutually recursive class definitions [GJS00, Str97]. (See also Section 4.4.)

1.2.3.5 Polymorphic Let

ML-style polymorphic let [Mil78, DM82] has the following typing rule, which cannot be modeled with the rules presented so far.

$$\frac{\Gamma \vdash e : s \quad \tilde{\alpha} = \mathbf{FV}(s) \setminus \mathbf{FV}(\Gamma) \quad \Gamma_x, x : \forall \tilde{\alpha}.s \vdash e' : t}{\Gamma \vdash \textbf{let } x = e \textbf{ in } e' : t}$$

Earlier approaches [CDDK86, Des84], including some recent ones [AF02], have chosen a direct rendering by a primitive operation $\mathtt{gen}(\Gamma, e)$ which computes just the desired variable set. However, this would require the `gen` operation to appear within the added rule at the third premise, thus mixing the TCG-level rule with the type-system-specific function `gen`. Furthermore, the operation appears somewhat ad-hoc, since it must be modified with the type system. For instance, Jones [Jon94, Figure 4] uses $\mathtt{gen}(\Gamma, P \Rightarrow \tau)$.

Inner Variables A cleaner approach is obtained by considering the intention of computing $\tilde{\alpha}$ as $\mathbf{FV}(e) \setminus \mathbf{FV}(\Gamma)$. We have that for any application of rule (let)

$$\Gamma \vdash e : s\sigma \quad \text{if } \mathit{dom}\, \sigma \subseteq \tilde{\alpha}$$

Hence, the value of e is indeed polymorphic, in that all instances of s can be derived as its type as well. A more detailed analysis is given in Section 2.4.1.2. This observation can be transferred to entire derivations. Consider the following application of the let rule, where

[13]Barendregt [Bar91] points out that abstraction and application can be re-iterated on the validity of kind expressions: Just as $t : *$ expresses that t is a well-formed type expression, $\kappa :: \Box$ expresses that κ is well-formed kind, with the axiom $* :: \Box$. Hence, the exact sorts and their axioms are *parameters* of a generalized type system.

$\mathcal{D}_0$, $\mathcal{D}$, $\mathcal{D}'$ and $\mathcal{D}''$ denote the derivations above these marks.

$$\dfrac{\mathcal{D}' : \Gamma \vdash e : s \qquad\qquad \mathcal{D}'' : \Gamma \cup \{\forall\tilde{v}[\,] \Longrightarrow x : s\} \vdash e' : s'}{\mathcal{D} : \Gamma \vdash \texttt{let}\ x = e\ \texttt{in}\ e' : s'}$$
$$\vdots$$
$$\mathcal{D}_0 : \Gamma_0 \vdash e_0 : t_0$$

In order to obtain arbitrarily many instances of $e : s\sigma$, we must instantiate the derivation $\mathcal{D}'\sigma$, without affecting the remainder of the derivation $\mathcal{D}_0$. This requires

$$dom\,\sigma \subseteq \mathbf{FV}(\mathcal{D}') \setminus \mathbf{FV}(\mathcal{D}_0\backslash_{\mathcal{D}'})$$

where $\mathcal{D}_0\backslash_{\mathcal{D}'}$ is the derivation tree $\mathcal{D}_0$ where the branch $\mathcal{D}'$ has been pruned. We call these variables the *inner variables* of derivation $\mathcal{D}'$. Let us write $\mathbf{IV}(\mathcal{D}', \mathcal{D}_0)$ for these. Since Γ is within $\mathcal{D}_0$ and outside of $\mathcal{D}'$, the proposed formalization is sound, that is no more variables than $\mathbf{FV}(e) \setminus \mathbf{FV}(\Gamma)$ will be generalized. For completeness, we have to show that Γ is indeed the only shared part between $\mathcal{D}'$ and $\mathcal{D}_0$. This result can, however, in principle not be shown for arbitrary derivations: The notion of derivation in Section 1.2.1.3 does not forbid renaming arbitrarily some variables in $\mathcal{D}'$, such that they coincide with variables from $\mathcal{D}'$. Hence, completeness can, as usual, be shown only with regard to a particular strategy of proof construction. We derive the necessary theorem in Section 2.4.6.

Subproof Extraction and Forward Resolution Besides determining the inner variables $\tilde{v}$, the rule $\forall\tilde{v}[\,] \Longrightarrow x : s$ must also bring together x and s in the conclusion. However, this cannot be achieved by actually writing down s outside of $\mathcal{D}'$, because that would cause the inner variables to become empty. Tcg therefore implements an atomic operation *subproof extraction* that converts the sub-derivation $\mathcal{D}'$ in a single step to a rule

$$\forall\big(\mathbf{IV}(\mathcal{D}', \mathcal{D}_0)\big)[\,] \Longrightarrow e : s$$

The desired rule is obtained through *forward resolution* [Pau94] with

$$\texttt{bind}_x = \forall(f, t)\big[f : t\big] \Longrightarrow x : t$$

By resolving the premise $f : t$ against the conclusion $e : s$, we obtain the new conclusion $x : s$, still quantified over the inner variables of $\mathcal{D}'$.

1.2.2 Remark. For an additional motivation, observe that the forward resolution with $\texttt{bind}_x$ corresponds to a cornerstone of the soundness proofs for (let), the substitution lemma (e.g. [Pie02, Lemma 9.3.8]) which reads:

If $\Gamma, x : s \vdash e : t$ and $\Gamma \vdash e' : s$, then $\Gamma \vdash e[e'/x]$.

The subproof of $\Gamma \vdash e : s$ from $\mathcal{D}'$ entails that the variable x, which is bound to e, can be given type t as well, because we can copy the subproof for e to any place where x is referenced.

The instructions to perform subproof extraction and forward resolution are embedded naturally into the context modifier $+r$ (Section 1.2.3.4), where r no longer a single rule, but a *rule expression* of the form

$$r ::= R \ \mid\ \texttt{fwd}(R, r) \ \mid\ \texttt{extract}(i)$$

where R is a rule and i is the number of the premise, whose derivation is to be extracted.

Extensions In Section 1.2.3.6 we will see that the introduced formulation generalizes to Jones' [Jon94] qualified types. Note also that the eigenvariable condition for quantifier introduction in sequent calculi (Section 1.2.2.2) uses a similar reasoning be requiring that the quantified variable is not free in the lower sequent of the rule instance.

Polymorphism and Recursion In Section 1.2.3.4 we have introduced the standard *letrec* rule as an instance of variable binding. The question arises naturally whether the treatment of polymorphism in this section carries over to recursive bindings – the computation $\mathbf{FV}(\mathcal{D}') \setminus \mathbf{FV}(\mathcal{D}_0)$ can only be performed after derivation $\mathcal{D}'$ is completely available. Polymorphic recursion, that is the availability of typing $x : \forall\tilde{\alpha}.s$ in the derivation of $e : s$ is in general undecidable, since semi-unification [KTU93] can be reduced to polymorphic recursion [Hen89b, Hen93]. We therefore do not attempt to model the semi-algorithms presented in [Hen89b].

1.2.3.6 Constraints

With syntax-directed typing rules, the construction of derivations directly parallels the term structure of the checked expression. Thus, the derivation is constructed by a straightforward recursive traversal of the syntax tree, and the main proof obligation consists in the compatibility of the involved types. Wand [Wan87] has introduced this distinction to the usual function application rule to obtain a simpler proof of type inference.

$$\frac{\Gamma \vdash f : t' \quad \Gamma \vdash e : s' \quad t' = s \to t,\ s = s'}{\Gamma \vdash (f\,e) : t}$$

Apart from the equality constraints, a derivation exists for every expression e, hence e is well-typed iff all equality constraints in its derivation are satisfied. For type inference, one chooses distinct type variables for each variable and each non-variable subexpression. The task then is to find a substitution for the type variables such that all the equalities are satisfied. Type inference thus becomes a two-phase process, with a straightforward recursion to the syntax tree and a subsequent unification to solve all equalities simultaneously. The two phases are commonly termed *constraint generation* and *constraint solution.*

The intuitive, and technical, appeal of the proceeding is that it decouples two essential tasks in type inference, the recursion through the source program and the treatment of relations between types. Unlike the tree-structured source code with bound variables, the constraints are (essentially) a flat set, which simplifies their treatment. The other way around, the main complexity of type inference is confined to constraint solution, and can be handled by specialized solvers [AFFS98, Pot01]. Furthermore, the interface between the tasks can be specified without reference to the source language: It is sufficient to know the structure of types and the language of constraints (e.g. [AFFS98, Sul00, SS01], see also Section 5.2).

Wand's approach has been generalized by Jones [Jon94]. Jones introduces *predicates* that can restrict the possible instances of type schemes. His applications are Haskell's type classes [WB89], extensible records and subtyping [Jon94, Section 3]. Unlike the equality constraints, predicates are possibly not solvable directly, hence they remain as open assumptions (in the sense of Section 1.2.2.1). Jones's formulation can be seen as switching from this natural deduction conception to sequent calculus conception (Section 1.2.2.2). He extends the judgments to contain all open predicates.

$$P \mid \Gamma \vdash e : t$$

The discharge of open assumptions (Section 1.2.2.2) is effected together with polymorphic generalization (Section 1.2.3.5). Hence, Jones considers type schemes of the form

$$\forall \tilde{\alpha}.P \Rightarrow t$$

This type scheme denotes all instance types $t' = \sigma(t)$, such that the predicates $\sigma(P)$ hold. More precisely, at the point where a value with that type scheme is used, all of its predicates must be entailed [Jon94, Section 3, Definition 2] by the predicates of the judgement (adapted from [Jon94, Figure 4]).

$$\frac{(x : \forall \tilde{\alpha}.P' \Rightarrow s) \in \Gamma \quad P'[\tilde{t}/\tilde{\alpha}] \Vdash P}{P \mid \Gamma \vdash x : s[\tilde{t}/\tilde{\alpha}]} \quad \text{(var)}$$

The polymorphic let then has the rule [Jon94, Figure 4]:

$$\frac{P \mid \Gamma \vdash e : s \quad P' \mid \Gamma_x, x : \forall \tilde{\alpha}.P \Rightarrow s \vdash e' : t \quad \tilde{\alpha} = (\mathbf{FV}(s) \cup \mathbf{FV}(P)) \setminus \mathbf{FV}(\Gamma)}{P' \mid \Gamma \vdash \mathbf{let}\ x = e\ \mathbf{in}\ e' : t} \quad (1.2.11)$$

Sulzmann, Odersky and Wehr [OSW99, Sul00] present a further generalization in the *HM*(X) framework. *HM*(X) extends the constraint language used for qualified types by existential quantification (projection on type variables). Starting with a non-empty set Ω of tokens (or predicates), the language is [Sul00, Section 3.1]:

$$C ::= \omega \mid C_1 \wedge C_2 \mid \exists \alpha C \qquad \omega \in \Omega$$

Sulzmann [Sul00] develops the theory of constraint systems and abstracts over the structure of tokens. He argues that Jones's discharge of open assumptions P in (1.2.11) is meaningful only in lazy languages, where e will be evaluated by need. If in rule (1.2.11) the set P of predicates is inconsistent (that is, its denotation is empty), the expression e should be considered erroneous, because it can never be used. Hence, (1.2.11) would allow erroneous subexpressions in a typable expression, as long as the bound variable x is never referred to. He proposes to use the following rule instead, where the discharged assumptions are kept in the conclusion with the intention of checking their satisfiability:

$$\frac{C \cup D, \Gamma \vdash e : \tau \quad \tilde{\alpha} \notin \mathbf{FV}(C) \cup \mathbf{FV}(\Gamma)}{C \cup \exists \tilde{\alpha}.D, \Gamma \vdash e : \forall \tilde{\alpha}.D \Rightarrow \tau} \quad (\forall\text{Intro})$$

Deferred Judgments Rather than introducing predicates P to judgments, TCG takes the natural deduction view of qualifications as open assumptions. Proof construction simply exempts some judgments from processing, and these are interpreted as constraints. We call these goals *deferred* to indicate that they must be resolved later on. The notation of qualification $P \Rightarrow \tau$ from (1.2.11) then reduces to operations on the proofs: The type assignment $x : \forall\tilde{\alpha}.P \Rightarrow \tau$ is identified with a rule $\forall\tilde{\alpha}\big[P\big] \Longrightarrow x : \tau$. Correspondingly, the judgement $P \mid \Gamma \vdash M : \tau$ is identified with a proof of $\Gamma \vdash M : \tau$ from deferred judgments P.

An immediate consequence is the direct integration with polymorphism by inner variables (Section 1.2.3.5): Since the deferred goals are part of the normal derivation, they are considered for quantification in just the same manner as the goals of the result type s of e are. Furthermore, the choice integrates with the representation of variable declarations by TCG rules (Section 1.2.3.2). Finally, deferred goals fit well into a framework with backward proof construction (Section 1.2.3.7).

Jones' entailment $\Vdash$ is a monotone, transitive relation that is closed under substitution [Jon94, Section 3]. The above formulation for TCG uses set inclusion, which obviously satisfies these axioms. Jones's remaining rules for λ abstraction and function application are also compatible with the interpretation of predicates as open assumptions, as they pass on the predicates P unmodified.

$$\frac{P \mid \Gamma \vdash f : s \to t \quad P \mid \Gamma \vdash e : s}{P \mid \Gamma \vdash f\,e : t}(\to\mathrm{E}) \qquad \frac{P \mid \Gamma_x, s : s \vdash e : t}{P \mid \Gamma \vdash \lambda x.e : s \to t}(\to\mathrm{I})$$

Instance: Coercions In imperative languages it is customary to change the representation of values if necessary. These transformations are called *conversions* (or *coercions*) (see Section 4.2.7). For example, a function application $f(e)$ would be translated to a internal form $f(c(e))$ with conversion c by

$$\frac{f : s \to t \quad e : s' \quad c : s' \leadsto s}{\Gamma \vdash f(e) \leadsto f(c(e)) : t}$$

The necessary conversion c can be determined only when both s' and s are known, which will eventually be the case since enough type annotations are provided by the programmer. The judgement $c : s' \leadsto s$ should therefore be deferred until the available type information has actually propagated. The alternative is, of course, to reason about the order of rule application, which may also require a re-formulation of rules. This second solution therefore contradicts the desire to provide declarative specifications of type systems.

1.2.3.7 Proof Construction

TCG's approach to proof construction is backward resolution of goals [Gal86]. Unlike Prolog, however, it chooses its goals from the local context of the judgments (Section 1.2.3.2). Therefore, the resolution steps is akin to the higher-order extension λProlog [Mil91, NM98]. TCG augments this strategy, however, with subproof extraction and forward application (Section 1.2.3.5). The lack of these features makes it hard to write a type inferencer for polymorphic let in λProlog [Lia97, Han98, Lia02].

The resolution of goals also does not proceed strictly from left to right: Some goals can be deferred (Section 1.2.3.6) for later processing and may remain as open assumptions. The method of selecting deferred goals is still under development. The current implementation uses syntactic predicates on terms, called *selectors* (Section 3.1.1). However, the TCG basic formalism as presented in Chapter 2 does not depend on the exact mechanism.

Tree Structure The TCG proof trees are constructed from the root to the leaves. Therefore, proof trees will be modeled explicitly as mappings from a tree domain [Cou83] to judgments. The special quality of proofs is the application of a suitable rule at each inner node, which is represented by an annotation function (see also [Gal86, Definition 3.4.5], [Ric78, Definition 4.1.7]).

States of Judgments The operation of rule extraction comes with a technical obstacle: The order in which the subtrees of the proof are constructed becomes relevant. In the (let)-example, the context of the second premise of the rule (let) is not available before the first premise has been completed. Hence, TCG proofs must allow judgments in five states:

pending A premise of a rule that does not yet have a corresponding node in the proof tree, because its context modifier could not yet be executed.
unresolved A goal to which no rule has yet been applied.
deferred A goal that is kept unsolved for later reference.
solved A judgement resolved by application of a rule.
discharged After rule extraction, the deferred judgments can be considered solved. Much of the development in Chapter 2 treats discharged judgments like deferred judgments, and discharged judgments are introduced in Section 2.5.1.

The stated demand on the order of resolution is directly parallel to the implementation of type checkers. Jones' checker for Haskell [Jon99, sec. 11.3] contains the following clauses:

```
tiExpr as (Ap e f)
  = do (ps,te) ← tiExpr as e
       (qs,tf) ← tiExpr as f
       t← newTVar Star
       unify (fn tf t) te
       return (ps ++ qs,t)
```

```
tiExpr as (Let bg e)
  = do (ps,as') ← tiBindGroup as bg
       (qs,t)   ← tiExpr (as'++as) e
       return (ps++qs,t)
```

Here *ps* and *qs* are lists of predicates [Jon94] for Haskell's type classes, the variables with prefix *t* are types, the *as* are type assumptions on the variables. The left clause checks function application by recursively checking the operator and operand positions independently. It combines the results by concatenating the predicates. The second clause checks the `let`-construct. It displays the dependency of the second premise on the result of the first premise by the transfer of as' (the type assumptions bindings in *bg*) between the premises.

Relation to Deferred Judgments The choice of backward resolution motivates the decision to refrain from introducing a set of predicates P to judgments in Section 1.2.3.6: If the constraints remain as normal judgments, they can still be processed by backward resolution. In Section 5.2 we will also investigate briefly the use of constraint handling rules (CHR) [Frü98] for that process.

Operational Semantics The formulation of the Prolog language starts from a declarative semantics of a program, from which its desired properties can be inferred (e.g. [HJ90]). Then, an operational semantics complements and implements the declarative one (e.g. [And92]). It would be straightforward to reformulate the Definition 2.3.3 of proofs as an inductive definition. However, the given one would still be needed to reason about incomplete proofs, as is done in Section 2.4.

1.2.3.8 Constraint Solution

We have seen in Section 1.2.3.6 how unsolved judgments can be integrated with Tcg's proof-based approach to type checking and type inference. However, Tcg does not currently support the *solution* of constraints beyond backward resolution (Section 1.2.3.7; it includes unification for syntactic equality constraints). The reason is simply that the constraint domains commonly considered in type inference require extensive separate treatments. Therefore, Tcg should be understood as a complementary framework: Given a constraint domain, it allows to specify a programming language and type system that generates the constraints. Note that this differs from, and generalizes, Jones' [Jon94] and Sulzmann's [Sul00] work (see also [AF02]), both of which investigate constraints in the setting of a particular language.

Subtyping Subtyping constraints were among the first constraint domains to be considered (e.g. [CW85, Mit91, AC93, EST95a]). The motivation for subtyping stems from modeling the inheritance relation in object-oriented languages (e.g. [CW85, EST95b, GM94, Lit98]), conversions [HR95], and their application to soft-typing (e.g. [Wri94, AWL94]).

The type checking problem for subtyping with recursive types is well-investigated and a relatively direct extension of type equality checking [AC93]. Also a formulation that can be input to Tcg is published [BH97]. However, this is not sufficient: As noted in Section 1.2.1.4, a minimal amount of type inference is necessary in practical programming languages. Tcg handles coercions and sub-class relations (Sections 4.3.3 and 4.4) as they occur in existing languages, thus solving subtyping constraints in special cases.

However, the full type inference problem with subtyping constraints is more complex, as it requires solving a system of general inequalities between types. Unlike Hindley-Milner type inference, it is in general not possible to find a substitution of types for type variables that solves all constraints. Instead, type inference checks for the *satisfiability* of constraint sets [AW93, EST95a, JP99, Pot01], which is sufficient to show soundness of the type system. This process requires closing the generated subtyping constraints under the rules

$$\frac{s \leq r}{s \leq t \quad t \leq r}(\text{trans}) \qquad \frac{s' \leq t' \quad t \leq t'}{s \to t \leq s' \to t'}(\text{decomp})$$

Note that the closure for (trans) requires multi-headed rules [Frü98] to match constraints $s \leq t$ and $t \leq r$ to produce $s \leq r$. The checked expression is considered well-typed if no unsatisfiable constraints arise. With this closure, the problem of type inference for subtyping is similar to that of data flow analysis [Hei92, FF96, FF99, NNH99]. The number of constraints in the closure grows exponentially, and dedicated mechanisms for handling them have to be devised [AFFS98, Pot01]. The main task is to *simplify* the closed constraint sets, that is to eliminate constraints without changing the denotation of the entire set.

1.2.3.9 Summary

In this Section 1.2.3, I have motivated the design of proofs in TCG. They will be presented formally in Chapter 2. I hope to have clarified that with this design, the typing features found in widely used programming languages can be expressed in TCG. A further investigation, under the more concrete premises of the given implementation, is found in Section 4.1.

I have furthermore examined the relation to conventional logical calculi in natural deduction (Section 1.2.2.1) and sequent style (Section 1.2.2.2). Although superficial similarities exist with both formalisms, none of them alone can capture the intended field of applications. In particular, the type inference for polymorphic *let* (Section 1.2.3.5) seems genuine to TCG.

Chapter 2

Proofs

Proofs are fundamental to TCG's conception of type checking. They represent both intermediate and final results, the atomic derivation steps are proof transformations, and complete proofs are interpreted as typing deductions in the formalized type discipline. This chapter formalizes TCG's notion of proof in Sections 2.2 and 2.3, building on term structures as introduced in Section 2.1. Section 2.4 defines the necessary atomic steps to extend partial proofs towards complete proofs. To establish the well-definedness of the steps, the proof structure is shown to be an invariant of the derivation process. Among the well-definedness results, the Instantiation Theorem 2.4.7 is a generalized version of lemmata found in conventional presentations of the ML type system (e.g. [Tof90, Lemma 4.2], [WF92, Lemma 4.5]). The final Section 2.5 adds straightforward extensions that facilitate the pratical application of TCG in Chapter 4.

2.1 Terms

The structure and definition of proofs in this chapter is largely independent of the terms used to encode the specific typing judgments. The derivations steps in Section 2.4 likewise depend only on a small set of atomic operations. Therefore, this section axiomatizes terms as abstract objects with a fixed set of operations.

The standard definition of terms as a freely generated set over function symbols and variables [Gal86, BS01] is an obvious instance of our term structure. Infinite regular trees [Cou83] with cyclic graph-unification [AHS86] fulfill the axioms as well. Higher-order patterns [Mil91, Nip93] with bound names can be integrated with minor modifications to the existing definitions. (See [Mil91, MNPS91, NM98] for integrating λ-terms with logic programming.)

2.1.1 Terms and Substitution

We study first the operation of substitution on terms with variables. The axioms both for application and composition are straightforward and we state the standard properties of these operations as lemmata.

2.1.1 DEFINITION (Terms). A *term structure with substitutions* is a tuple

$$\mathcal{T} = \langle \mathcal{T}, \mathcal{V}, \overset{\mathcal{T}}{=}, \mathbf{FV}, @_{\mathcal{S}}, \circ \rangle$$

where $\mathcal{T}$ and $\mathcal{V}$ are countable sets with $\mathcal{V} \subseteq \mathcal{T}$[1], $\mathbf{FV}\colon \mathcal{T} \to \mathcal{P}(\mathcal{V})$ and, $\circ\colon \mathcal{S} \times \mathcal{S} \to \mathcal{S}$ with $\mathcal{S} := \mathcal{V} \overset{\text{fin}}{\to} \mathcal{T}$, $@_{\mathcal{S}} : \mathcal{T} \times \mathcal{S} \to \mathcal{T}$. $\mathcal{T}$ are the *terms*, $\overset{\mathcal{T}}{=}$ is *equality* and **FV** are the *free variables*. $\mathcal{S}$ are the *substitutions*, $@_{\mathcal{S}}$ is *application of substitutions*, the function $\circ$ is *composition of substitutions*. We write the substitutions as $\{v_1 \mapsto t_1 \mathinner{..} v_n \mapsto t_n\}$. The elements $v_i \mapsto t_i$ are the *bindings* of the substitution. We canonically extend **FV** and $\overset{\mathcal{T}}{=}$ to sets of terms. **FV** is extended to substitutions as

$$\mathbf{FV}(\sigma) := \mathbf{FV}(\mathit{range}(\sigma)) \cup \mathit{dom}(\sigma)$$

A term structure satisfies the following equalities for substitutions σ, σ_1, σ_2, σ_3, terms t, and variable v.

$$v \mapsto t \in \sigma \implies t \neq v \tag{2.1.1}$$

$$t@_{\mathcal{S}}\varnothing \overset{\mathcal{T}}{=} t \tag{2.1.2}$$

$$v@_{\mathcal{S}}\sigma \overset{\mathcal{T}}{=} \begin{cases} \sigma(v) & \text{if } v \in \mathit{dom}(\sigma) \\ v & \text{otherwise} \end{cases} \tag{2.1.3}$$

$$t@_{\mathcal{S}}\sigma \overset{\mathcal{T}}{=} t@_{\mathcal{S}}\sigma|_{\mathbf{FV}(t)} \tag{2.1.4}$$

$$(\sigma_1 \circ \sigma_2) \circ \sigma_3 \overset{\mathcal{T}}{=} \sigma_1 \circ (\sigma_2 \circ \sigma_3) \tag{2.1.5}$$

$$t@_{\mathcal{S}}(\sigma \circ \tau) \overset{\mathcal{T}}{=} t@_{\mathcal{S}}\tau@_{\mathcal{S}}\sigma \tag{2.1.6}$$

$$\mathbf{FV}(t@_{\mathcal{S}}\sigma) \overset{\mathcal{T}}{=} \mathbf{FV}(t) \setminus \mathit{dom}(\sigma) \cup \mathbf{FV}(\mathit{range}(\sigma|_{\mathbf{FV}(t)})) \tag{2.1.7}$$

By (2.1.2) $\mathbf{1} := \varnothing$ is the *identity substitution.*

2.1.2 NOTATION. We write $t\sigma$ and $\sigma(t)$ as short forms for $t@_{\mathcal{S}}\sigma$ in view of (2.1.3). By (2.1.5), σ^n is the n-fold composition of σ with itself.

We will write $s = t$ for $s \overset{\mathcal{T}}{=} t$, because $\overset{\mathcal{T}}{=}$ is the only equality needed on terms (including variables).

The first lemma about term structures shows that the equational characterization of composition relative to application makes that operation the same as in conventional presentations [Rob65, Definition 5.5].[2]

2.1.3 LEMMA. $\sigma \circ \tau = \sigma|_{\complement\, \mathit{dom}(\tau)} \cup \{v \mapsto s \mid v \in \mathit{dom}(\tau),\ s = v@_{\mathcal{S}}\tau@_{\mathcal{S}}\sigma \neq v\}$

Proof. Let $\phi := \sigma \circ \tau$ and $\phi' = \sigma|_{\complement\, \mathit{dom}(\tau)} \cup \{v \mapsto s \mid v \in \mathit{dom}(\tau),\ s = v@_{\mathcal{S}}\tau@_{\mathcal{S}}\sigma \neq v\}$. By definition of substitutions as functions, two substitutions are equal if they agree on every $v \in \mathcal{V}$. We proceed by case distinction. If $v \notin \mathit{dom}(\sigma) \cup \mathit{dom}(\tau)$, then

$$\phi(v) \overset{(2.1.3)}{=} v@_{\mathcal{S}}\phi \overset{(2.1.6)}{=} v@_{\mathcal{S}}\tau@_{\mathcal{S}}\sigma \overset{(2.1.4),\,(2.1.2)}{=} v = \phi'(v)$$

[1] In a generic implementation of term structures an embedding function is necessary to adapt the representation [Gas01, Appendix A.2].

[2] $\complement\cdot$ denotes set complement.

If $v \in dom(\sigma)$, $v \notin dom(\tau)$ then by the same reasoning $\phi(v) = v@_S\sigma = \sigma(v) = \phi'(v)$. If $v \in dom(\tau)$, then $\phi(v) = v@_S\tau@_S\sigma$, but by (2.1.1) this component must not be included if it is the identity. Hence, we have $\phi(v) = \phi'(v)$. •

2.1.4 Corollary. *A substitution σ is* idempotent[3] *iff* $\mathbf{FV}(range(\sigma)) \cap dom(\sigma) = \varnothing$.

2.1.5 Corollary. *If σ, τ are substitutions such that $dom\,\sigma$, $range\,\sigma$, $dom\,\tau$ and $range\,\tau$ are pairwise disjoint, then $\sigma \circ \tau = \tau \circ \sigma = \sigma \cup \tau$.*

2.1.6 Definition. A term t *matches* a term p, if there is a substitution ϕ with $\phi(p) = t$. The matching relation is written as $p \leq^{\phi} t$, or $p \leq t$ if the substitution is not relevant. Define the relation *rigid match* by

$$p \leq_r t := \begin{cases} \text{True} & p \leq t \\ \text{False} & p \not\leq t \wedge \nexists \tau . p \leq t\tau \\ \downarrow & \text{otherwise} \end{cases}$$

We also say that p is (not) matched by t rigidly.

2.1.7 Remark. The name *rigid* indicates an analogy with rigid heads in higher-order unification [Hue75]. If $p \not\leq_r t$ then there is no way for substitutions to change this relation, just as rigid heads cannot be changed by unification to allow a unifier.

2.1.2 Unifiers and the Generalization Order

Unification is the basic procedure on terms needed by the derivation steps (Section 2.4). We therefore define the notion of *terms with most general unifiers* as a property of term structures. A second, derived operation is the extension of unification to substitutions, which leads to least upper bounds on (idempotent) substitutions. Eder [Ede85] gives a detailed account on the necessary prerequisites for such an operation, albeit for freely generated terms. His study carries over to our term structures and we trace the main results. Palamidessi [Pal90] extends Eder's work to an algebraic theory of substitutions; his motivation is a clear semantics for parallel execution of Prolog programs.

2.1.8 Definition (More general substitutions). Let σ and τ be substitutions. *τ is more general than σ* (or equivalently *σ is more special than τ*), if there is a substitution ϕ with $\phi \circ \tau = \sigma$. This relation is written $\tau \leq^{\phi} \sigma$, and ϕ can be omitted where it is irrelevant. The two substitutions are *equivalent* (written $\sigma \sim \tau$) if both $\sigma \leq \tau$ and $\tau \leq \sigma$.

2.1.9 Lemma. *The relation $\leq$ is a pre-order on $\mathcal{S}$ by associativity (2.1.5).*

2.1.10 Definition (Most General Unifiers). A substitution σ is a *unifier* of a finite set S of terms iff $s\sigma = t\sigma$ for all $s, t \in S$. A unifier σ is a *most general unifier* for a finite set S of terms if $\sigma \leq \sigma'$ for any unifier σ' of S. A term structure *has most general unifiers* iff for any set S of terms that does have a unifier, S has a most general unifier.

By extension, σ is a (most general) unifier of a finite set of finite sets of terms iff it is a (most general) unifier of each of the sets. It is a (most general) unifier of the pair $\langle s, t\rangle$ iff it is a (most general) unifier of $\{s, t\}$, and likewise for the equation $s = t$. σ is a (most general) unifier of a set of equations if it is a (most general) unifier of each of them.

[3]that is $\sigma^2 = \sigma$

2.1.11 Remark. If σ is a most general unifier of S, then $dom\,\sigma \subseteq \mathbf{FV}(S)$, because any binding $v \mapsto r \in \sigma$, $v \notin \mathbf{FV}(S)$ can be removed without affecting $S\sigma$ by (2.1.4).

The $\sim$-relation between equivalent substitutions (Definition 2.1.8) can be characterized by the notion of renamings, which are injective substitutions into the variables.

2.1.12 Definition (Renaming). A substitution ρ is a *renaming* if $range(\rho) \subseteq \mathcal{V}$ and there is a ρ^{-1} with $\rho \circ \rho^{-1} = \rho^{-1} \circ \rho = \mathbf{1}$.

2.1.13 Lemma ([Ede85, Lemma 2.10]). *For equivalent substitutions $\sigma \sim \tau$, there are renamings ρ, ρ' such that $\tau = \rho \circ \sigma$, $\sigma = \rho' \circ \tau$, and $\rho' = \rho^{-1}$.*

Proof. Eder's proof is formulated in terms of substitutions as total functions that are the identity almost everywhere; the main ideas carry over to the setting of Definition 2.1.1, where substitutions are finite functions. By assumption, $\tau = \rho \circ \sigma$ and $\sigma = \rho' \circ \tau$ for some ρ, ρ'; we show that these substitutions have the claimed properties. First, by assumption $\sigma = \rho' \circ (\rho \circ \sigma)$, such that for all $x \in \mathbf{FV}(range(\sigma))$ $\rho'(\rho(x)) = x$; as $\rho(x) \in \mathbf{FV}(range(\rho \circ \sigma)) = \mathbf{FV}(range(\tau))$, ρ is an injective function $\mathbf{FV}(range(\sigma)) \to \mathbf{FV}(range(\tau))$. Because these sets are finite, it follows that $|\mathbf{FV}(range(\sigma))| \leq |\mathbf{FV}(range(\tau))|$. By symmetry, also $\rho' : \mathbf{FV}(range(\tau)) \to \mathbf{FV}(range(\sigma))$ and $|\mathbf{FV}(range(\tau))| \leq |\mathbf{FV}(range(\sigma))|$. Hence, $|\mathbf{FV}(range(\tau))| = |\mathbf{FV}(range(\sigma))|$ and both ρ and ρ' are renamings and inverse to each other. ●

2.1.14 Convention. In the remainder of this thesis, we identify equivalent unifiers and

$$\sigma = mgu(S)$$

denotes that σ is a most general unifier of set of terms S (or set of set of terms, or set of equations, respectively) if it exists. Among the equivalent unifiers of S, *mgu* chooses an arbitrary one. If S does not have a unifier, then $mgu(S) :=\downarrow$.

The starting point for Eder's discussion is that a (finite) set of substitutions that has an upper bound with respect to $\lessdot$ does not necessarily have a supremum. This property holds, however, for the idempotent substitutions modulo $\sim$ [Ede85, Theorem 4.9]. The main point of the proof is that idempotent substitutions can be characterized (up to renamings) as unifiers of finite sets of finite sets of terms [Ede85, Proposition 4.5]. Obviously, an idempotent substitution $\sigma = \{x_i \mapsto t_i\}_{t=1}^n$ is a unifier of $\{\{x_i, t_i\}\}_{i=1}^n$. Conversely, the idempotent substitutions are exactly the unifiers of finite sets of finite sets.

2.1.15 Lemma ([Ede85, Proposition 4.5]). *If σ is a substitution, then (a) and (b) are equivalent.*

(a) *There is a finite set of finite sets M such that σ is a most general unifier of M.*
(b) *There is an idempotent substitution σ' which is equivalent to σ.*

2.1.16 Lemma ([Ede85, Lemma 4.6]). *If M and N are sets of sets of terms and σ is a most general unifier of M, then*

(a) *$M \cup N$ is unifiable iff $N\sigma$ is unifiable.*
(b) *If τ is a most general unifier of $N\sigma$, then $\tau \circ \sigma$ is a most general unifier of $M \cup N$.*

2.1.17 Lemma ([Ede85, Proposition 4.8]). *Let M and N be two finite sets of finite sets of terms. Then*

$$mgu(M \cup N) = \sup\{mgu(M), mgu(N)\} \ .$$

Eder concludes [Ede85, Theorem 4.9] that the idempotent substitutions modulo $\sim$ form a lattice with smallest and greatest element. More important for the later application in Tcg is a consequence of the previous Lemma that Eder mentions: It allows the computation of the supremum (with respect to $\lessdot$, up to equivalence $\sim$) of a pair of substitutions.

2.1.18 Theorem. *Let σ and τ be idempotent substitutions in a term structure with most general unifiers. The supremum of σ and τ can be determined by unification, if it exists:*

$$\sigma \sqcup \tau := \big(mgu(\ \{\{x, \tau(x)\} \mid x \in dom(\tau)\}\sigma)\big) \circ \sigma$$

If the supremum does not exist, we write $\sigma \sqcup \tau = \downarrow$.

Proof. Define $M = \{\{x, \sigma(x)\} \mid x \in dom(\sigma)\}$ and $N := \{\{x, \tau(x)\} \mid x \in dom(\tau)\}\}$. Obviously, $\sigma \in mgu(M)$ and $\tau \in mgu(N)$. Therefore,

$$\begin{aligned} \sigma \sqcup \tau = \sup(mgu(M), mgu(N)) &\overset{\text{Lemma 2.1.17}}{=} mgu(M \cup N) \\ &\overset{\text{Lemma 2.1.16}}{=} mgu(\sigma N) \circ \sigma = mgu(\sigma\{\{x, \tau(x)\} \mid x \in dom(\tau)\}) \circ \sigma \end{aligned}$$

•

2.1.3 Fixed Term Structure

In the remainder of this chapter, we fix a term structure $\mathcal{T}$ with most general unifiers and speak of *the* terms, substitutions, etc. although it is understood that these notions remain parameters of the calculus and only the properties of Definition 2.1.1 are used. In examples, we freely choose a suitable term language with function symbols and predicates found in the respective type theories.

2.2 Rules, Contexts and Judgments

The definition of a rule must be inductive according to the analysis in Section 1.2: When a rule is applied, it may modify the context in which its premises are to be proven, including the addition of rules to this context. The definition can be structured further by two auxiliary notions, *context modifiers* and *rule expressions*. Both of them serve as indirections to the point in the proof where a rule is applied (Section 2.3).

2.2.1 Definition. The sets *Rule*, *Rule expression* and *Context modifier* and the relation *references* are defined inductively. The induction base is given by $n = 0$ in cases 1 and 3.

1. Let $v_1 \,..\, v_m$ be variables, $a_0 \,..\, a_n$ terms, and $M_1 \,..\, M_n$ context modifiers, such that if a rule expression in M_i references j, then $j < i$. Then the following is a rule:

   $$\forall v_1 \,..\, v_m \big[M_1 \vdash a_1 \,..\, M_n \vdash a_n\big] \Longrightarrow a_0$$

   The rule is *quantified over $v_1 \,..\, v_m$*; if $m = 0$ then the quantifier $\forall$ may be omitted. The components $M_i \vdash a_i$ are the *premises* of the rule, a_0 is its *conclusion*.

2. A *rule expression* has one of the forms

$$\begin{array}{rll} RE ::= & \mathit{Rule} & \textit{basic rule} \\ \mid & \mathcal{R}^{\forall}(\mathrm{i}) & \textit{extract and quantify} \\ \mid & \mathcal{R}^{f}(\mathit{Rule},\mathit{RE}) & \textit{forward reasoning} \end{array}$$

   A rule expression r *references* j if r contains a subexpression $\mathcal{R}^{\forall}(j)$. Define the set $\mathcal{R}^{\mathrm{ref}}_{\forall}(r) := \{i \mid \mathcal{R}^{\forall}(i) \text{ is a subexpression of } r\}$.
3. A *context modifier* is a sequence $M = m_1 \mathinner{\ldotp\ldotp} m_n$ where for all $i = 1 \mathinner{\ldotp\ldotp} n$ either $m_i = +r$ with a rule expression r or $m_i = -\exists V.t$ with a set of variables V and term t. By extension, a context modifier M *references* j if M contains some rule expression that references j. Define also $\mathcal{R}^{\mathrm{ref}}_{\forall}(M) := \bigcup_{+r\in M} \mathcal{R}^{\forall}(r)$.

A *context* is a set of rules.

The definitions of the free variables and substitution are extended to rules, context modifiers and contexts in the obvious manner. The only notable point concerns the treatment of quantification: The quantified variables in rules are considered bound (in the usual sense) in the rule. However, the rule expression $\mathcal{R}^{\forall}$ does not introduce any bound variables: The binding takes place only when the rule expression is evaluated during rule application (Definition 2.2.11). This construction leads to variable capture, which is desirable in the intended application of the polymorphic `let` (Section 4.1.3.3).

2.2.2 Definition. The free variables of rules, rule expressions and context modifiers are defined by structural induction over the rule expressions:

$$\begin{aligned}
\mathbf{FV}\big(\forall V\,[M_1 \vdash a_1 \mathinner{\ldotp\ldotp} M_n \vdash a_n] \Longrightarrow a_0\big) :=& \left(\mathbf{FV}(a_0) \cup \bigcup_{i=1}^{n} \mathbf{FV}(M_i) \cup \mathbf{FV}(a_i)\right) \setminus V \\
\mathbf{FV}(r) :=& \begin{cases} \mathbf{FV}(R) & \text{if } r = R \text{ is a rule} \\ \varnothing & \text{if } r = \mathcal{R}^{\forall}(i) \\ \mathbf{FV}(R) \cup \mathbf{FV}(r') & \text{if } r = \mathcal{R}^{f}(R, r') \end{cases} \\
\mathbf{FV}\big(\langle m_1 \mathinner{\ldotp\ldotp} m_n\rangle\big) :=& \bigcup_{i=1}^{n} \mathbf{FV}(m_i) \\
\mathbf{FV}(+r) :=& \mathbf{FV}(r) \\
\mathbf{FV}(-\exists V.t) :=& \mathbf{FV}(t) \setminus V
\end{aligned}$$

The bound variables are defined symmetrically:

$$\begin{aligned}
\mathbf{BV}\big(\forall V\,[M_1 \vdash a_1 \mathinner{\ldotp\ldotp} M_n \vdash a_n] \Longrightarrow a_0\big) :=& V \cup \bigcup_{i=0}^{n} \mathbf{BV}(M_i) \\
\mathbf{BV}(r) :=& \begin{cases} \mathbf{BV}(R) & \text{if } r = R \text{ is a rule} \\ \varnothing & \text{if } E = \mathcal{R}^{\forall}(i) \\ \mathbf{BV}(R) \cup \mathbf{BV}(r') & \text{if } E = \mathcal{R}^{f}(R, r') \end{cases} \\
\mathbf{BV}\big(\langle m_1 \mathinner{\ldotp\ldotp} m_n\rangle\big) :=& \bigcup_{i=1}^{n} \mathbf{BV}(m_i) \\
\mathbf{BV}(+r) :=& \mathbf{BV}(r) \\
\mathbf{BV}(-\exists V.t) :=& V
\end{aligned}$$

Both **FV** and **BV** are extended to contexts by set-union.

2.2.3 Convention. We adopt the Barendregt variable convention [Bar84, Conv. 2.1.13]: Given a set of objects currently under consideration, their bound variables are disjoint from their free variables, and the sets of bound variables at different quantifiers are pairwise disjoint. Any collisions are resolved by silently renaming the offending bound variables.

2.2.4 Definition. Application of substitutions is extended to rules and context modifiers.

$$\begin{aligned}
\sigma\big(\forall V\big[M_1 \vdash a_1 \mathinner{..} M_n \vdash a_n\big] \Longrightarrow a_0\big) &:= \forall V\big[\sigma(M_1) \vdash a_1\sigma \mathinner{..} \sigma(M_n) \vdash a_n\sigma\big] \Longrightarrow a_0\sigma \\
\sigma(r) &:= \begin{cases} \sigma(R) & \text{if } r = R \text{ a rule} \\ r & \text{if } r = \mathcal{R}^{\forall}(i) \\ \mathcal{R}^{f}(\sigma(R), \sigma(r')) & \text{if } r = \mathcal{R}^{f}(R, r') \end{cases} \\
\sigma\big(\langle s_1 \mathinner{..} s_n\rangle\big) &:= \langle \sigma(s_1) \mathinner{..} \sigma(s_n)\rangle \\
\sigma(+r) &:= +\sigma(r) \\
\sigma(-\exists V.t) &:= -\exists V.\sigma(t)
\end{aligned}$$

For a context Γ, the application is

$$\sigma(\Gamma) := \{\sigma(r) \mid r \in \Gamma\}$$

The properties of free variables and substitution from Definition 2.1.1 carry over to rules, because application to rules is a homomorphic extension of the basic application to terms.

2.2.5 Lemma. *Let R be a rule, and let σ, τ be substitutions.*

$$\begin{aligned}
R\mathbf{1} &= R \\
R\sigma &= R\sigma|_{\mathbf{FV}(R)} \\
R(\sigma \circ \tau) &= R\tau\sigma \\
\mathbf{FV}(R\sigma) &= \mathbf{FV}(R) \setminus \mathit{dom}(\sigma) \cup \mathbf{FV}(\mathit{range}(\sigma|_{\mathbf{FV}(R)}))
\end{aligned}$$

Proof. By structural induction and Definitions 2.2.2 and 2.2.4, using Definition 2.1.1 for the base case. •

2.2.6 Definition. A *judgement* is a proposition $\Gamma \vdash a$ with a context Γ and a term a. Γ is also called *the context of the judgment.* The set of judgments is denoted by $\mathcal{J}$. The free and bound variables of a judgement are

$$\mathbf{FV}(\Gamma \vdash a) := \mathbf{FV}(\Gamma) \cup \mathbf{FV}(a) \qquad \mathbf{BV}(\Gamma \vdash a) := \mathbf{BV}(\Gamma)$$

Substitutions are applied to judgments component-wise:

$$\sigma\big(\Gamma \vdash a\big) = \sigma(\Gamma) \vdash \sigma(a)$$

2.2.1 Execution of Context Modifiers

The evaluation of rule expressions (Definition 2.2.11) requires the operation of *forward resolution* [Pau94]. Forward resolution combines two rules into a single rule, such that resolution of a goal with that rule is the same as using the two input rules in sequence. For example, let R and R' be rules as in Definition 2.2.7 below. They could be applied in sequence to obtain the following proof:

$$\cfrac{\cfrac{M'_1(M_1(\Gamma)) \vdash p'_1 \mathinner{.\,.} M'_{n'}(M_1(\Gamma)) \vdash p'_{n'}}{M_1(\Gamma) \vdash p_1}\,R' \qquad M_2(\Gamma) \vdash p_2 \mathinner{.\,.} M_n(\Gamma) \vdash p_n}{\Gamma \vdash c}\,R \tag{2.2.1}$$

This example motivates the following definition:

2.2.7 Definition (Forward Resolution). Let two rules

$$\begin{aligned} R &= \forall V\big[M_1 \vdash p_1 \mathinner{.\,.} M_n \vdash p_n\big] \Longrightarrow c \\ R' &= \forall V'\big[M'_1 \vdash p'_1 \mathinner{.\,.} M'_{n'} \vdash p'_{n'}\big] \Longrightarrow c' \end{aligned}$$

be given such that

$$\begin{gathered} n > 0 \\ \sigma = mgu(p_1, c') \neq\downarrow \\ dom(\sigma) \subseteq V \cup V' \end{gathered} \tag{2.2.2}$$

Then $fwd\_resolve(R, R')$ yields the rule

$$\forall V, V'\big(\big[M_1 \cdot M'_1 \vdash p'_1 \mathinner{.\,.} M_1 \cdot M'_{n'} \vdash p'_{n'},\ M_2 \vdash p_2 \mathinner{.\,.} M_n \vdash p_n\big] \Longrightarrow c\big)\sigma \tag{2.2.3}$$

otherwise $fwd\_resolve(R, R') =\downarrow$.

2.2.8 Remark. Condition (2.2.2) ensures that forward resolution remains a local function that does not induce a replacement of free variables (Section 2.4.1). Such replacements would be undesirable, because the order of the executed context modifiers (Definition 2.2.13) would affect the set of result rules, while the user has only limited possibilities to influence or specify the order. The examples in Chapter 4 use forward resolution with a rule R whose premises are closed and linear, and rule R' represents some completely established fact, which does not need to be modified any further. Thus, the condition does not restrict the applicability of the construct.

2.2.9 Remark. The result (2.2.3) does not contain new free variables, because the result rule is re-quantified over the bound variables of the input rules.

2.2.10 Remark. The correspondence of $fwd\_resolve(R, R')$ with the motivation (2.2.1) is not perfect, because the rules applicable at a judgement are chosen from the judgment's context (Definition 2.3.3). Thus, if M_1 removes R' from Γ, then the proof (2.2.1) cannot be constructed, while the forward resolution step forces R' to be applied.

A context modifier M is executed on a context Γ by executing the elements of M sequentially. Because the rule expression $\mathcal{R}^\forall(\cdot)$ is intended to refer to subtrees of the proof under construction, they must be resolved through an indirection, which resembles the assignment of values to variables in interpretations of formulae (e.g. [Gal86, Sections 3.3 and 5.3]). The subproof environment Φ in the following definition thus will contain the characteristic parts of subproofs in Definition 2.3.3.

2.2.11 Definition (Evaluation of rule expressions). Let Φ be a *subproof environment*

$$\Phi : \mathbb{N} \overset{\text{fin}}{\to} \mathcal{P}(\mathcal{V}) \times (\mathcal{J}^* \times \mathcal{J})$$

Define the auxiliary function

$$\text{AsRule}(\langle \Gamma_i \vdash p_i \rangle_{i=1}^m, \Gamma \vdash c) := \big[\vdash p_1 \mathinner{..} \vdash p_m\big] \Longrightarrow c$$

For a rule expression r, the *evaluation of r under Φ*, written r_Φ, is defined if $\mathcal{R}_\forall^{\text{ref}}(r) \subseteq dom(\Phi)$. Then r_Φ is given by

$$\begin{aligned} R_\Phi &= \{R\} \\ \mathcal{R}^\forall(i)_\Phi &= \{\forall I.\text{AsRule}(S)\} && \text{where } \Phi(i) = \langle I, S\rangle \\ \mathcal{R}^f(R,r)_\Phi &= \{R'' \mid R' \in r_\Phi, R'' = \mathit{fwd\_resolve}(R,R') \neq \downarrow\} \end{aligned}$$

2.2.12 Remark. Evaluation of a rule expression according to Definition 2.2.11 yields at most one rule as a result. The formulation in terms of sets of rules avoids making evaluation a partial function. In this manner, *fwd_resolve* can act as a filter for newly introduced rules. Furthermore, in Section 2.5 we will extend $\mathcal{R}^\forall$ to yield a set of subproofs, and this extension is anticipated in the above definition.

2.2.13 Definition (Context modification). Let Φ be a subproof environment as in Definition 2.2.11. The *execution of a context modifier M* on a context Δ is defined by:

$$\begin{aligned} \Delta @^\Phi \varepsilon &:= \Delta \\ \Delta @^\Phi \langle m_1 \mathinner{..} m_n\rangle &:= \Delta' @^\Phi \langle m_2 \mathinner{..} m_n \rangle \\ \text{where } \Delta' &= \begin{cases} \Delta \cup r_\Phi & m_1 = +r \\ \Delta \setminus \{R \mid R = (\forall \tilde{v}\big[P\big] \Longrightarrow c) \in \Delta \text{ and } t \trianglelefteq_r c\} & m_1 = -\exists V.t \end{cases} \end{aligned}$$

2.2.14 Remark. The matching in removal is rigid (Definition 2.1.6) to ensure that later substitutions to c cannot invalidate an executed removal operation (Section 2.4.1): The match, hence the entire execution of the modifier, is undefined if any instance of c would require the rule to be removed, even though it is not removed with the present conclusion c. This proviso will be essential in Section 2.4.1.1.

2.3 Proofs

A proof, as motivated in Section 1.2, is a tree labeled with judgments (Definition 2.2.6) that represent type checking judgments. The tree structure is expressed as a mapping from a tree-domain to judgments [Cou83, Gal86], which enables the representation of incomplete proofs during backwards construction (Section 2.4). The main property of proofs, as opposed to general trees, is that a node is related to its children by an applicable rule.

This constraint is expressed by an annotation function, which attaches to each node the applied rule. We proceed in two steps: Definition 2.3.1 considers only the tree structure. Definition 2.3.3 adds the requirement that each non-leaf judgement in a proof tree can in fact be proven by a rule from its context.

2.3.1 Definition. Let t be a tree over judgments (Definition 2.2.6). Its free and bound variables, and application of substitutions are defined by extension:

$$\mathbf{FV}(t) := \bigcup_{p \in dom(t)} \mathbf{FV}(t[p]) \qquad \mathbf{BV}(t) := \bigcup_{p \in dom(t)} \mathbf{BV}(t[p]) \qquad t\sigma := \{p \mapsto t[p]\sigma\}_{p \in dom(t)}\,.$$

The *inner variables* of the subtree $t\{p\}$ are those variables not occurring elsewhere in t.

$$\mathbf{IV}(t,p) := \mathbf{FV}(t\{p\}) \setminus \mathbf{FV}(t\backslash_p)$$

2.3.2 Remark. The inner variables reflect a requirement from Section 1.2.3.5: To implement polymorphic let [Mil78, DM82, CDDK86], those variables must be identified that can be replaced independently for each reference to a polymorphic value. These are precisely the inner variables (see Sections 2.4.1.2, 2.4.6 and 4.1.3.3).

2.3.3 Definition. A *pre-proof* is either $\perp$ (*failure*) or a triple $P = \langle t, f, F \rangle$ where t is a tree labeled with judgments and F is a set of *fixed variables.* The function f maps tree positions in t to *annotations*

$$f\colon \mathbb{N}^* \to \{\text{Deferred}\}\dot{\cup}\mathit{Rule}$$

and the $dom(f)$ is a tree domain.

A subtree $t\{p\}$ is *complete* if $dom(t\{p\}) \subseteq dom(f)$ and for all $pq \in dom(f)$, where $f(pq)$ is a rule with n premises, we have $pqi \in dom(t)$ for $i = 1 \mathinner{..} n$.

The *subproof environment in t at pi* (where $p \neq \varepsilon$) is

$$\Phi_t^{pi} := \big\langle \mathbf{IV}(t,pj), \langle D_j, t[pj]\rangle \big\rangle_{j=1}^{i-1}$$
$$\text{where } D_j = \langle t[l] \mid l \in \text{leaves}(t\{pj\}), f(l) = \text{Deferred}\rangle$$

A pre-proof P is a *proof* if the following conditions hold:

1. $dom(f) \subseteq dom(t)$.
2. If $p \in dom(t) \setminus dom(f)$, then p is a leaf in t.
3. If $t[p] = \Gamma \vdash a$ and

$$r := f(p) = \forall V \big[M_1 \vdash d_1 \mathinner{..} M_n \vdash d_n \big] \Longrightarrow c$$

then there is a substitution σ with $dom(\sigma) \subseteq V$, $\mathbf{FV}(range(\sigma)) \cap V = \varnothing$ such that all of the following hold:

   (a) $r \in \Gamma$
   (b) $a = c\sigma$
   (c) If $pi \in dom(t)$ then
      i. $i \in 1 \mathinner{..} n$ and $t[pi] = \Gamma_i \vdash d_i\sigma$ and $\Gamma_i \subseteq \Gamma @^{\Phi_t^{pi}}(M_i\sigma)$.
      ii. for $j \in \mathcal{R}_\forall^{\text{ref}}(M_i)$ the subtree $t\{pj\}$ is complete.
      iii. for $j \in \mathcal{R}_\forall^{\text{ref}}(M_i)$ we have $\mathbf{IV}(t,pj) \subseteq F$.

2.3.4 Remark. Conditions 1 and 2 ensure that the proof is tree-structured and the annotation function is defined only on existing tree nodes. Condition 2 adds that all inner nodes of t are annotated.

The main property that distinguishes proofs from pre-proofs is that each node is related to its children by an applied rule. It is formalized in 3. Condition 3a requires the applied rule to be chosen from the local context of the application. 3b and 3(c)i require that the judgments in the proof are instances of the conclusion and premises of the applied rule, and that the context modifiers are obeyed. Condition 3(c)iii notes those variables that have once been observed as inner variables of some subproof as fixed, thus inhibiting further substitutions involving these variables (Definition 2.3.9). Condition 3(c)ii ensures that only complete subproofs can be extracted, such that after extraction, the tree structure of the proofs is fixed as well.

2.3.5 Definition. We use the notation from Definition 2.3.3. A judgemnet at p in t is called *solved* iff f is defined at p; an unsolved leaf is a *goal* and

$$\text{goals}\,\big\langle t, f, F\big\rangle = \{g \mid g \in \mathit{dom}(t) \setminus \mathit{dom}(f)\}$$

A proof $P = \big\langle t, f, F\big\rangle$ is *complete* if $t[\varepsilon]$ is complete according to Definition 2.3.3.

The root judgement $t[\varepsilon]$ is also called the *initial judgement* of P, and we say that P is *a proof of* $t[\varepsilon]$. The context of the root judgement is called the *initial context.* For $p \in \mathit{dom}(f)$, $f(p) = r$ is a rule, we say that r *is is applied to judgement* $t[p]$ or r *is applied at* p.

2.3.6 Example (Context Modification). The workings of context modification are illustrated in rule ($\rightarrow$-intro) for λ-abstraction:

$$\frac{\Gamma_x \cup \{x : s\} \vdash e : t}{\Gamma \vdash \lambda x.e : s \rightarrow t}$$

The Tcg-formalization is (Section 4.1.1, see Section 3.1.3 for the form of context modifiers):

$$\forall\,(x,e,s,t)\ \frac{-\,(:_1 = x);\ +\,\big[x : s\big] \vdash e : t}{\lambda x.e : (s \rightarrow t)}\ (\texttt{lambda})$$

Suppose there is a λ-abstraction to be type-checked at tree node

$$t[p] = \Gamma \vdash \lambda x'.e' : s' \rightarrow t'$$

with $f(p) = \texttt{lambda}$. According to 3c, the first and only child is

$$t[p1] = \Gamma' \vdash e'' : s''\ .$$

According to 3b and 3c we must have $e' = e''$ and $s' = s''$, for otherwise a matching σ (Condition 3) could not be found. Furthermore, $\sigma(x) = x'$ and $\sigma(s) = s'$. We are interested in $\Gamma' \subseteq \Gamma @^{\varnothing}(M\sigma)$. From Definition 2.2.13 we see that

$$\begin{aligned}\Gamma @^{\varnothing}(M\sigma) &= (\Gamma \setminus \{\big[\tilde{M}\big] \Longrightarrow x\sigma : u \mid u \text{ any term}\}) \cup \{[\,] \Longrightarrow x\sigma : s\sigma\}\\ &= (\Gamma \setminus \{\big[\tilde{M}\big] \Longrightarrow x' : u \mid u \text{ any term}\}) \cup \{[\,] \Longrightarrow x' : s'\}\end{aligned}$$

2.3.7 Remark (State of Judgments). Section 1.2.3.7 postulated four states *pending*, *unresolved*, *solved*, and *deferred* for judgments and a fixed order in which these states are entered. The states are encoded by the interplay of the functions t and f: A pending judgement for premise i at p is not present in the proof tree at all, that is $t[pi]$ is undefined. A non-pending, but unresolved judgement has $t[pi]$ defined, but $f(pi)$ is undefined. A solved judgement can either be resolved by a rule or deferred for later processing, depending on the $f(pi)$.

The order of insertion of pending judgments to t is determined already by the property that both $dom(f)$ and $dom(t)$ are tree domains [Gal86, Section 2.2] and Condition 1. This restriction enforces that premises of rules are inserted to the tree t left-to-right. Likewise, the cross-references between subtrees by rule expressions $\mathcal{R}_{\forall}^{\mathrm{ref}}$ allow only backward references according to Definition 2.2.1.

2.3.8 Remark (Weakening). Condition 3(c)i could be simplified to

$$t[pi] = \Gamma @^{\Phi}(M_i\sigma) \vdash d_i\sigma\,.$$

The simplification concerns the context of $t[pi]$, which is to be equal to the context of $t[p]$, modified according to the rule. Looking at Condition 3a, this is overly restrictive: The applicable rules are chosen from the context and adding more rules does not invalidate a proof. In other words, the chosen form 3(c)i introduces *weakening* (or *monotonicity*) (e.g. [Gen35, RS92]):

$$\frac{\Gamma \vdash a}{\Gamma \cup \Delta \vdash a}$$

The chosen form also integrates more smoothly with the consistency requirements of derivation steps (see Remark 2.4.8).

The interaction of context modification with application of a substitution $\sigma(P)$ deserves special attention. If $P = \langle t, f, F\rangle$ is a proof, we expect $P' = \sigma(P)$ to be a proof as well. The details will be considered in Section 2.4.1. The main proviso is that substitutions must be compatible with P in the following sense.

2.3.9 Definition (Compatible Substitution). Let $P = \langle t, f, F\rangle$ be a proof and let τ be a substitution. τ is *compatible with* P if $\mathbf{FV}(\tau) \cap F = \varnothing$.

2.3.10 Definition. Let $P = \langle t, f, F\rangle$ be a proof and σ a substitution compatible with P. Then $\sigma(P) := \langle t', f', F'\rangle$ where

$$\begin{aligned} t' =& \{p \mapsto t[p]\sigma \mid p \in dom(t)\} \\ f' =& \{p \mapsto r\sigma \mid f(p) = r \in \mathit{Rule}\} \\ & \cup \{p \mapsto a \mid f(p) = a \notin \mathit{Rule}\} \end{aligned}$$

We also write $P\sigma := \sigma(P)$.

2.4 Derivation Steps

This section demonstrates that the proof construction steps implemented in Chapter 3 can be carried out such that the proof structure of Sections 2.2 and 2.3 is maintained. In this way, the conditions on proofs stated in Definition 2.3.3 are invariants on the proof process as motivated in Section 1.2.

2.4.1 Instantiation

At several points during proof construction, a proof P must be instantiated to a proof $P' = \sigma(P)$. We are going to show that P' is a proof if σ is compatible with P (Theorem 2.4.7). As proofs represent type deductions, this result is more than a technical necessity: Polymorphic let, which can be implemented by subproof extraction (Section 4.1.3.3), requires in the soundness proofs a lemma of the form (see for example [Tof90, Lemma 4.2], [WF92, Lemma 4.5])

$$\Gamma \vdash e : t \qquad \text{implies} \qquad \Gamma\sigma \vdash e : t\sigma \tag{2.4.1}$$

This lemma is an instance of Theorem 2.4.7, if we interpret complete Tcg proofs as type derivations via some interpretation function $\mathcal{I}_P$:

$$\begin{tikzcd} P \arrow[r, "\text{Theorem 2.4.7}"] \arrow[d, "\mathcal{I}_P"'] & P\sigma \arrow[d, "\mathcal{I}_P"] \\ \mathcal{I}_P(P) \arrow[r, "(*)"] & \mathcal{I}_P(P\sigma) \end{tikzcd}$$

The inference (*) is usually proven by reconstructing a typing derivation for the instantiated judgement in (2.4.1). With Theorem 2.4.7 that derivation is given by interpreting the instantiated proof $P\sigma$, whose root judgement is just the instantiated root judgement of the original P, and that root judgments is interpreted as right-hand-side judgement in (2.4.1). A closer comparison can be found in Section 2.4.1.2.

2.4.1.1 Instantiation Theorem

The instantiation theorem is proven in three steps, arguing in each one that application of substitutions do not destroy the invariants on proofs expressed in Definition 2.3.3. First, we consider the effect of substitutions on rule expressions. Second, we prove that context modification is not affected. Third, we get the desired instantiation theorem.

2.4.1 Notation. For the remainder of this Section 2.4.1.1, we use the notation of Condition 3, Definition 2.3.3 and Definition 2.4.1: $P = \big\langle t, f, F \big\rangle$ is a proof and τ is an idempotent substitution compatible with P. $t[p]$ is a judgement to which the rule

$$\forall V \big[M_1 \vdash d_1 \mathinner{.\,.} M_n \vdash d_n \big] \Longrightarrow c$$

has been applied. σ is the local substitution of Condition 3 and pi is a path in $dom(t)$ as in Condition 3c. $j \in \mathcal{R}^{\text{ref}}_{\forall}(M_i)$ is any of the references. The corresponding entities in $P' = P\sigma$ are denoted by t', M_i', d_i', c', σ'. As usual, local fresh definitions of these names override the ones above.

2.4.2 Lemma. *Substitution does not influence the inner variables of extracted subproofs. For any $q \in dom(t)$ such that*

$$\mathbf{IV}(t,q) \subseteq F \tag{2.4.2}$$

we have

$$\mathbf{IV}(t\tau,q) = \mathbf{IV}(t,q)$$

Proof. First, we abbreviate the mentioned subtrees by:

$$A := t\{q\} \qquad B := t\backslash_q$$

Let us state two consequences of the precondition that τ is compatible with P and (2.4.2)

$$\mathbf{IV}(t,q) \cap (dom(\tau) \cup \mathbf{FV}(range(\tau))) = \varnothing$$

hence (by $A \cap B = \varnothing \implies A \cap \complement B = A$)

$$\mathbf{IV}(t,q) \cap \complement\, dom(\tau) \cap \complement\mathbf{FV}(range(\tau|_{\mathbf{FV}(B)})) = \mathbf{IV}(t,q) \tag{2.4.3}$$

A second consequence from

$$dom(\tau) \cap \mathbf{IV}(t,q) = dom(\tau) \cap (\mathbf{FV}(A) \setminus \mathbf{FV}(B)) = \varnothing \tag{2.4.4}$$

is that τ can never introduce variables into B which it does not introduce into A at the same time. Hence,

$$\tau|_{\mathbf{FV}(A)} = \tau|_{\mathbf{FV}(A)\cap(\mathbf{FV}(B)\cup\complement\mathbf{FV}(B))} \overset{(2.4.4)}{=} \tau|_{\mathbf{FV}(A)\cap\mathbf{FV}(B)}$$

and therefore

$$\begin{aligned}\mathbf{FV}(range(\tau|_{\mathbf{FV}(A)})) &\setminus \mathbf{FV}(range(\tau|_{\mathbf{FV}(B)})) \\ &= \mathbf{FV}(range(\tau|_{\mathbf{FV}(A\cap B)})) \setminus \mathbf{FV}(range(\tau|_{\mathbf{FV}(B)})) = \varnothing\end{aligned} \tag{2.4.5}$$

With these two observations, we can compute (using idempotence of τ at $(*)$):

$$\begin{aligned}
\mathbf{IV}(t\tau,q) =& \mathbf{FV}(A\tau) \setminus \mathbf{FV}(B\tau) \\
(2.1.7)\quad =& \big(\mathbf{FV}(A) \setminus dom(\tau) \cup \mathbf{FV}(range(\tau|_{\mathbf{FV}(A)}))\big)\setminus \\
& \big(\mathbf{FV}(B) \setminus dom(\tau) \cup \mathbf{FV}(range(\tau|_{\mathbf{FV}(B)}))\big) \\
=& \big((\mathbf{FV}(A) \cap \complement\, dom(\tau)) \cup \mathbf{FV}(range(\tau|_{\mathbf{FV}(A)}))\big) \\
& \cap \big((\complement\mathbf{FV}(B) \cup dom(\tau)) \cap \complement\mathbf{FV}(range(\tau|_{\mathbf{FV}(B)}))\big) \\
=& (\mathbf{FV}(A) \cap \underline{\complement\, dom(\tau))} \cap \big((\complement\mathbf{FV}(B) \cup \underline{dom(\tau))} \cap \complement\mathbf{FV}(range(\tau|_{\mathbf{FV}(B)}))\big) \\
(*)\quad & \cup\underline{\mathbf{FV}(range(\tau|_{\mathbf{FV}(A)}))} \cap \big((\complement\mathbf{FV}(B) \cup \underline{dom(\tau))} \cap \complement\mathbf{FV}(range(\tau|_{\mathbf{FV}(B)}))\big) \\
=& \mathbf{FV}(A) \cap \complement\, dom(\tau) \cap \complement\mathbf{FV}(B) \cap \complement\mathbf{FV}(range(\tau|_{\mathbf{FV}(B)})) \\
& \cup\mathbf{FV}(range(\tau|_{\mathbf{FV}(A)})) \cap \complement\mathbf{FV}(range(\tau|_{\mathbf{FV}(B)})) \cap \complement\mathbf{FV}(B) \\
=& (\mathbf{FV}(A) \setminus \mathbf{FV}(B)) \cap \complement\, dom(\tau) \cap \complement\mathbf{FV}(range(\tau|_{\mathbf{FV}(B)})) \\
& \cup\mathbf{FV}(range(\tau|_{\mathbf{FV}(A)})) \cap \complement\mathbf{FV}(range(\tau|_{\mathbf{FV}(B)})) \cap \complement\mathbf{FV}(B) \\
=& \mathbf{IV}(t,q) \cap \complement\, dom(\tau) \cap \complement\mathbf{FV}(range(\tau|_{\mathbf{FV}(B)})) \\
& \cup\mathbf{FV}(range(\tau|_{\mathbf{FV}(A)})) \cap \complement\mathbf{FV}(range(\tau|_{\mathbf{FV}(B)})) \cap \complement\mathbf{FV}(B) \\
(2.4.3),\ (2.4.5)\quad =& \mathbf{IV}(t,q)
\end{aligned}$$

•

2.4.3 LEMMA. *Formation of subproof environments commutes with substitution.*

$$(\Phi_t^{pi})\tau = \Phi_{t\tau}^{pi}$$

Proof. Immediate from the Definition 2.3.3 and Lemma 2.4.2. •

2.4.4 LEMMA. *Forward resolution commutes with substitution. Let R, R' be two rules as in Definition 2.2.7.*

$$\text{If } \mathit{fwd\_resolve}(R, R') \neq\downarrow \text{ then } \mathit{fwd\_resolve}(R\tau, R'\tau) = (\mathit{fwd\_resolve}(R, R'))\tau$$

If $\mathit{fwd\_resolve}(R, R') =\downarrow$ then $\mathit{fwd\_resolve}(R\tau, R'\tau)$ may be defined.

Proof. The first part is direct by Definition 2.2.7, the required substitution is

$$\sigma' := \{v \mapsto \tau(\sigma(v))\}_{v\in dom(\sigma)} = (\tau \circ \sigma)|_{dom(\sigma)}$$

By Convention 2.2.3, $\mathbf{FV}(\tau) \cap (V \cup V') = \varnothing$, and by Definition 2.2.7 $dom(\sigma) = dom(\sigma') \subseteq V \cup V'$, hence $p_1\tau\sigma' = c'\tau\sigma'$ by Corollary 2.1.5.

The second statement follows from restriction (2.2.2): If unification of p_1 with c' requires instantiating some free variables, thus failing (2.2.2), the substitution τ may contain just the required instantiation. •

2.4.5 LEMMA. *Evaluation of the rule expressions commutes with instantiation.*

$$(r\tau)_{\Phi_{t\tau}^{pi}} \supseteq (r_{\Phi_t^{pi}})\tau$$

Proof. The claim is proved by a direct structural induction on r, using Definition 2.2.13. Let $\Phi := \Phi_t^{pi}$ and $\Phi' := \Phi_{t\tau}^{pi}$.

$\underline{r = R \in \mathit{Rule}}$ $(R_\Phi)\tau = R\tau = (R\tau)_{\Phi'}$

$\underline{r = \mathcal{R}^\forall(i)}$ Let $\Phi(i) = \langle V, S\rangle$, $\Phi'(i) = \langle V', S'\rangle$.

$$\begin{aligned}(\mathcal{R}^\forall(i))\tau_\Phi &\overset{\text{Def. 2.2.11}}{=} (\forall V.\text{AsRule}(S))\tau \\ &\overset{\text{Def. 2.3.9}}{=} \forall V.(\text{AsRule}(S)\tau) \overset{\text{Def. 2.2.11}}{=} \forall V.\text{AsRule}(S\tau) \\ &\overset{\text{Lemma 2.4.2}}{=} \forall V'.\text{AsRule}(S') = \mathcal{R}^\forall(i)_{\Phi'} \overset{\text{Def. 2.2.4}}{=} (\mathcal{R}^\forall(i)\tau)_{\Phi'}\end{aligned}$$

$\underline{r = \mathcal{R}^f(R, r')}$ By Lemma 2.4.4. Note that the set-inclusion can be strict. •

2.4.6 LEMMA. *Let Γ and Γ' be a contexts and M a context modifier.*

$$\Gamma' \supseteq \Gamma\tau \implies \Gamma'@^{\Phi_{t\tau}^{pi}}M\tau \supseteq \big(\Gamma@^{\Phi_t^{pi}}M\big)\tau \tag{IH}$$

Proof. By induction on the length of M. The case $M = \varepsilon$ is clear.

$\underline{M = \langle +r\rangle \cdot \hat{M}}$ Let $\Phi := \Phi_t^{pi}$ and $\Phi' := \Phi_{t\tau}^{pi}$. Assume $\Gamma' \supseteq \Gamma\tau$. To make the induction more visible, define

$$\Delta' := \Gamma' \cup (r\tau)_{\Phi'} \qquad \Delta := \Gamma \cup r_\Phi$$

By Lemma 2.4.5 and assumption, we have $\Delta' \supseteq \Delta\tau$. Then conclude, using Definition 2.2.13

$$\begin{aligned}\Gamma'@^{\Phi'}M\tau &= (\Gamma' \cup (r\tau)_{\Phi'})@^{\Phi'}\hat{M}\tau \\ &= \Delta'@^{\Phi'}\hat{M}\tau \overset{\text{IH}}{\supseteq} \big(\Delta@^{\Phi}\hat{M}\big)\tau \\ &= \big((\Gamma \cup r_\Phi)@^{\Phi}\hat{M}\big)\tau = \big(\Gamma@^{\Phi}M\big)\tau\end{aligned}$$

<u>$M = \langle -\exists V.t \rangle \cdot \hat{M}$</u> As in the previous case, define two contexts

$$\begin{aligned}
\Delta &= \Gamma \setminus \{R \mid R = (\forall U\big[P\big] \Longrightarrow c) \in \Gamma, t \mathrel{\dot{\leq}_r} c\} \\
\Delta' &= \Gamma' \setminus \{R \mid R = (\forall U\big[P\big] \Longrightarrow c)\tau \in \Gamma', t\tau \mathrel{\dot{\leq}_r} c\tau\} \\
&\overset{\dagger}{=} \Gamma' \setminus \{R \mid R = (\forall U\big[P\big] \Longrightarrow c)\tau \in \Gamma', t\tau \mathrel{\dot{\leq}_r} c\}
\end{aligned}$$

The equality (†) is by Definition 2.1.6 of rigid matching: If the instantiation of c were to induce $t \mathrel{\dot{\leq}} c$, then $t \mathrel{\dot{\leq}_r} c$ is undefined. Hence from assumption $\Gamma' \supseteq \Gamma\tau$ we have $\Delta' \supseteq \Delta\tau$, and we can proceed by induction as in the previous case. •

2.4.7 Theorem. *The set of proofs is closed under application of substitution. For any proof P and substitution τ compatible with P*

$$P\tau = \big\langle t\tau, f\tau, F \big\rangle$$

is a proof.

Proof. We check the compliance of $P\tau$ with Definition 2.3.3. Conditions 1 and 2 are clearly satisfied. It remains to check Condition 3 on the correct application of rules. We use the notation from the definition. By definition $r\tau \in \Gamma\tau$, validating 3a. As the required substitution, use

$$\sigma' := (\tau \circ \sigma)|_V$$

where σ is the existing substitution for P. This cxhoice validates Condition 3b. By Convention 2.2.3, we have

$$dom\,\tau \cap V = \varnothing \tag{2.4.6}$$
$$\mathbf{FV}(range\,\tau) \cap V = \varnothing \tag{2.4.7}$$

Hence

$$a\tau \overset{P \text{ proof}}{=} c\sigma\tau \overset{\text{Lemma 2.1.3}}{=} c(\underbrace{\tau|_{\complement\, dom\,\sigma}}_{\overset{(2.4.6)}{=}\tau} \cup \underbrace{\{v \mapsto v\sigma\tau \mid v \in dom\,\sigma, v\sigma\tau \neq v\}}_{=\sigma'} \overset{(2.4.7)}{=} c\tau\sigma'$$

For Condition 3(c)i, use a similar reasoning and Lemma 2.4.6 (with $\Gamma' = \Gamma\tau$) for

$$\Gamma_i\tau = \overset{P \text{ proof}}{\subseteq} \big(\Gamma @^{\Phi} M_i\sigma\big)\tau \overset{\text{Lemma 2.4.6}}{\subseteq} \Gamma\tau @^{\Phi'} M_i\tau$$

Condition 3(c)ii is clear. Condition 3(c)iii follows from Lemma 2.4.2. •

2.4.8 Remark. The proofs of Lemma 2.4.5 and 2.4.6 show at two places that weakening (Remark 2.3.8) appears naturally: In Lemma 2.4.6, the pattern for removal may be instantiated such that fewer terms are removed. In Lemma 2.4.5 the forward resolution may deliver more result rules. The applications in Chapter 4 do not depend on this behaviour, their type systems are formalized such that in both places a further instantiation is not possible, either because the terms are ground already, or because they contain only bound variables.

2.4.1.2 Relation to the Instantiation Lemma

The fixed variables F of a proof $\langle t, f, F\rangle$ have a structural counterpart in conventional presentations of type systems, where they occur in the proofs of the counterpart of Theorem 2.4.7. For instance, Tofte [Tof90] proves the following lemma:[4]

LEMMA ([Tof90, lemma 4.2]). *If* $\Gamma \vdash e : t$ *and* σ *is a substitution, then* $\Gamma\sigma \vdash e : t\sigma$.

The proof of the lemma is by induction on the structure of e (or equivalently on the structure of the typing derivation, because the typing rules are syntax-directed). All cases are straightforward, except for the (let)-rule being applied as the last rule in the derivation. The (let)-rule employs a closure operation to compute the variables to be generalized.

$$\mathrm{Clos}_\Gamma t := \forall\tilde{\alpha}.t \text{ where } \tilde{\alpha} = \mathbf{FV}(t) \setminus \mathbf{FV}(\Gamma)\ .$$

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma\dot{\cup}\{x \mapsto \mathrm{Clos}_\Gamma t_1\} \vdash e_2 : t}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : t} \tag{let}$$

The induction step constructs a new derivation of $\Gamma\sigma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : t\sigma$ from derivations of specialized premises. Unfortunately, it is not sufficient to apply the induction hypothesis to the first premise: The result $\Gamma\sigma \vdash e_1 : t_1\sigma$ is correct, but it is not useful to prove the second premise. For that premise, the induction hypothesis yields

$$\big(\Gamma\dot{\cup}\{x \mapsto \mathrm{Clos}_\Gamma t_1\}\big)\sigma \vdash e_2 : t\sigma$$

but to obtain the structure of the (let)-rule, we need

$$\mathrm{Clos}_\Gamma(t_1\sigma)\ .$$

Note that the very proof obligation of Lemma 2.4.5 arises here again: The application of σ and the Clos operation must commute. This is not the case in general, because σ may substitute terms for the type variables in $\mathbf{FV}(t) \setminus \mathbf{FV}(\Gamma)$, thus influencing the closure.

Tofte's proof continues by restricting σ *locally*, that is only for the derivation step under consideration. Let $\mathrm{Clos}_\Gamma t_1 = \forall\tilde{\alpha}.t_1$ and choose a renaming $\rho = \{\alpha_i \mapsto \beta_i\}_{i=1}^n$ with fresh β_i. Then define

$$\sigma' = \sigma|_{\mathbf{FV}(\Gamma)} \cup \rho \tag{2.4.8}$$

Since the α_i do not occur in Γ by definition of Clos, we have $\Gamma\sigma' = \Gamma\sigma$ and so by the induction hypothesis

$$\Gamma\sigma \vdash e_1 : t_1\sigma'\ . \tag{†}$$

[4]The original statement is in terms of a type environment TE, substitution S, expression e and type τ. Tofte writes $\mathrm{TE} \rightarrow^S \mathrm{TE}'$ [Tof90, Definition 4.1], because due to renamings of bound variables, the application of substitution is not a function but a relation. The lemma is then: “If $\mathrm{TE} \vdash e \Rightarrow \tau$ and $\mathrm{TE} \rightarrow^S \mathrm{TE}'$ then $\mathrm{TE}' \vdash e \Rightarrow S\tau$.” We have simplified that statement to $\mathrm{TE}' = \sigma(\mathrm{TE})$, since by Convention 2.2.3 the distinction is not relevant to the discussion.

Now, the restricted substitution σ' and Clos do commute in the desired sense:

$$\begin{aligned}
\sigma(\mathrm{Clos}_\Gamma t_1) &= \sigma(\forall\tilde{\alpha}.t_1) \\
&= \forall\tilde{\beta}.t_1(\sigma|_{\mathbf{FV}(t_1)\setminus\tilde{\alpha}} \cup \rho) && \text{by definition of substitution and binding} \\
&= \forall\tilde{\beta}.t_1(\sigma|_{\mathbf{FV}(\Gamma)} \cup \rho) && \mathbf{FV}(t_1)\setminus(\mathbf{FV}(t_1)\setminus\mathbf{FV}(\Gamma)) = \mathbf{FV}(t_1)\cap\mathbf{FV}(\Gamma) \\
&= \forall\tilde{\beta}.t_1\sigma' = \mathrm{Clos}_{\Gamma\sigma}t_1\sigma'
\end{aligned}$$

For the last equality, say $\mathrm{Clos}_{\Gamma\sigma}t_1\sigma' = \forall\tilde{\gamma}.t_1\sigma'$ with $\tilde{\gamma} = \mathbf{FV}(t_1\sigma')\setminus\mathbf{FV}(\Gamma\sigma)$. Now argue that $\tilde{\gamma} = \tilde{\beta}$. Since none of the β_i is free in $\Gamma\sigma$, but does occur in $t_1\sigma'$ (α_i is free in t_1), we have $\tilde{\gamma} \supseteq \tilde{\beta}$. Conversely, assume there is a $\gamma_j \notin \tilde{\beta}$. Since $\gamma_j \in \mathbf{FV}(t_1\sigma')$, there must be some $\delta \in \mathbf{FV}(t_1)$ with $\gamma_j \in \mathbf{FV}(\delta\sigma')$ (possibly $\sigma'(\delta) = \delta$) with $\delta \in \mathbf{FV}(\Gamma)$ (if $\delta \notin \mathbf{FV}(\Gamma)$ then $\delta \in \tilde{\alpha}$). But with $\gamma_j \in \mathbf{FV}(\delta\sigma')$ and $\delta \in \mathbf{FV}(\Gamma)$, also $\gamma_j \in \mathbf{FV}(\Gamma\sigma')$, contrary to our assumption.

$$\begin{gathered}
\sigma(\Gamma \cup \{x \mapsto \mathrm{Clos}_\Gamma t_1\}) = \Gamma\sigma \cup \{x \mapsto \mathrm{Clos}_{\Gamma\sigma}t_1\sigma'\} \\
\Gamma\sigma \cup \{x \mapsto \mathrm{Clos}_{\Gamma\sigma}t_1\sigma'\} \vdash e_2 : t\sigma \qquad (\ddagger)
\end{gathered}$$

An application of rule (let) to the two premises (†) and (‡) yields conclusion $\Gamma\sigma \vdash e : t\sigma$.

Relation to Fixed Variables The structural correspondence with the fixed variables of Definition 2.3.3 is exhibited clearly in the choice of σ' in (2.4.8): σ' acts as a renaming on the fixed variables and behaves like σ on the remaining variables. Consequently, σ can only replace freely the non-fixed variables, which is just the condition of *compatibility* (Definition 2.3.9) in our definitions. Contrary to our setting σ is arbitrary in the lemma itself. The reason is that Tofte can discard the derivation $\Gamma\sigma \vdash e : t\sigma$ entirely and reconstruct the derivation of $\Gamma\sigma \vdash e : t\sigma'$. In other words, the substitutions for $\tilde{\alpha}$, which we disallow, are immaterial in Tofte's setting, only because he does not need to preserve the previous proof. Our fixed variables overcome the necessity of rebuilding the proof structure.

2.4.2 Resolution

Define for proof $P = \langle t, f, F\rangle$ a partial function

$$\langle t, f, F\rangle \xrightarrow[R,p]{\text{Resolve}} \langle t', f', F'\rangle$$

with preconditions

1. $p \notin dom(f)$
2. $t[p] = \Gamma \vdash a$
3. $R = \forall V\big[M_1 \vdash d_1 \mathinner{..} M_n \vdash d_n\big] \Longrightarrow c \in \Gamma$
4. ρ is a renaming with $dom(\rho) = V$, $range(\rho) \cap \mathbf{FV}(P) = \varnothing$.
5. $\sigma = mgu(a, c\rho) \neq\downarrow$ with $\mathbf{FV}(\sigma) \cap F = \varnothing$.

Then define

$$t' := t\sigma \qquad f' := (f \cup \{p \mapsto R\})\sigma \qquad F' := F$$

2.4.9 LEMMA. *If P is a proof and $P \xrightarrow[R,p]{Resolve} P' \neq\downarrow$ then P' is a proof.*

Proof. Since σ is compatible with P, by Theorem 2.4.7, $P\sigma$ is a proof. The extension of f at p satisfies Condition 3 of Definition 2.3.3. •

2.4.10 REMARK. Precondition 5 does not restrict the useful resolution steps that are possible on P, since $\mathbf{FV}(a, R) \cap F \neq \varnothing$ is never forced by the given definitions.[5] Consider Definition 2.3.3, that could force variables into F. Since $p \notin dom(f)$, no subproof containing p can have been extracted by Condition 3(c)ii, hence Condition 3(c)iii does not apply. Furthermore, no inner variables that are fixed by in Condition 3(c)iii are removed from their subproof, since Definition 2.2.11 quantifies the extracted rules over those variables. Hence, neither goal a nor rule R can contain fixed variables from this source.

2.4.3 Adding a Pending Goal

Define for proof $P = \langle t, f, F \rangle$ a partial function

$$\langle t, f, F \rangle \xrightarrow[pi]{\text{Add}} \langle t', f, F' \rangle$$

with preconditions

1. $f(p) = \forall V \left[M_1 \vdash d_1 \mathinner{..} M_n \vdash d_n \right] \Longrightarrow c$
2. for $j = 1 \mathinner{..} i - 1$: $pj \in dom(t)$
3. If $j \in \mathcal{R}_{\forall}^{\text{ref}}(M_i)$ then $t\{pj\}$ is complete.

Let σ be the substitution of Condition 3 in Definition 2.3.3. Then define

$$t' := t \cup \{pi \mapsto \Gamma @^{\Phi_t^{pi}} M_i\sigma \vdash d_i\sigma\}$$

and

$$F' := F \cup \bigcup_{k \in \mathcal{R}_{\forall}^{\text{ref}}(M_i)} \mathbf{IV}(t, pk)$$

2.4.11 LEMMA. *If $P = \langle t, f, \sigma \rangle$ is a proof and $P \xrightarrow[pi]{Add} P' \neq\downarrow$ then P' is a proof.*

Proof. Check the conditions of Definition 2.3.3. Conditions 1 and 2 are clear. It remains to check 3, which holds by the preconditions and definition of the operation. •

2.4.4 Deferring a Goal

For proof $\langle t, f, F \rangle$ define the partial function

$$\langle t, f, F \rangle \xrightarrow[p]{\text{Defer}} \langle t, f', F \rangle$$

under the precondition

$$p \in dom(t) \setminus dom(f)$$

Then define

$$f' := f \cup \{p \mapsto \text{DEFERRED}\}$$

[5]Although, of course, F can be enlarged arbitarily without contradicting the definitions.

2.4.12 LEMMA. *If P is a proof and $P \xrightarrow[p]{\mathit{Defer}} P' \neq\downarrow$ then P' is a proof.*

Proof. Immediate from Definition 2.3.3. •

2.4.5 Grafting Proofs

The implementation of the branch-expand mechanism (Section 3.3.4.5) combines independently constructed proofs. This operation is justified by the following definitions.

Let $P = \langle t, f, F\rangle$ and $P' = \langle t', f', F'\rangle$ be proofs. Define the partial function

$$P \leftharpoonup_p P' := \langle t'', f'', F''\rangle$$

if the following preconditions are met

1. $p \in dom(t) \setminus dom(f)$
2. $\rho : \mathbf{FV}(P') \setminus F' \to \mathcal{V} \setminus \mathbf{FV}(P)$ a renaming.
3. σ is a most general substitution such that $t[p]\sigma = t'[\varepsilon]\rho\sigma$ and $\mathbf{FV}(\sigma) \cap (F \cup F') = \varnothing$.

Then define

$$\begin{aligned} t'' &:= t\sigma|_{\complement p} \cup \{pq \mapsto J\rho\sigma \mid q \mapsto J \in t''\} \\ f'' &:= f\sigma|_{\complement p} \cup \{pq \mapsto a\rho\sigma \mid q \mapsto a \in f''\} \\ F'' &:= F \cup F' \end{aligned}$$

2.4.13 LEMMA. *Let P and P' be proofs. If $P \leftharpoonup_p P' = P'' \neq\downarrow$ then P'' is a proof.*

Proof. Check the conditions in Definition 2.3.3. Conditions 1 and 2 are fulfilled, because P and P' are proofs. Condition 3 is clear for all the inner nodes except for p and its parent node, because all of the sub-conditions are local and relative to the proof nodes. (Condition 3(c)ii with Precondition 1 ensures that $t\{p\}$ has not been extracted before.) For the node p, observe that the entire structure of P' has been copied there, replacing the original node $t[p]$. For the parent of p, use the Precondition 3. •

2.4.14 REMARK. The computation of the substitution σ in Precondition 3 is prohibitively expensive, because it requires set unification [Sto99] on the context Γ. It can be avoided if the derivation steps keep track of each application of a substitution and P' is constructed starting from the judgement $t[p]$ itself. Then we have

$$\langle\{\varepsilon \mapsto t[p]\}, \varnothing, F\rangle \to^* P'$$

In this situation, σ is just the substitution accumulated through the inference steps. The procedure can be extended to several independent judgments $k = 1 \mathinner{..} n$ with

$$\langle\{\varepsilon \mapsto t[p_k]\}, \varnothing, F\rangle \to^* P'_k$$

If this yields substitutions $\sigma_1 \mathinner{..} \sigma_n$, then the sought substitution is $\sigma = \sigma_1 \sqcup \cdots \sqcup \sigma_n$ by Theorem 2.1.18 (for idempotent substitutions). Section 3.3.1.1 shows how this computation can be accomplished efficiently.

2.4.6 Outer Variables

The proof structure defined in Definition 2.3.3 is sufficient to interpret the proofs constructed by TCG as typing derivations in Chapter 4, which ensures soundness of the generated type checkers. One further invariant on the proof construction process is necessary to demonstrate completeness of polymorphic let by subproof extraction (Section 4.1.3.3). Its typing rule is [Mil78, DM82, CDDK86]:

$$\frac{\Gamma \vdash e : s \quad \tilde{\alpha} = \mathbf{FV}(s) \setminus \mathbf{FV}(\Gamma) \quad \Gamma, x : \forall \tilde{\alpha}.s \vdash e' : t}{\Gamma \vdash \mathbf{let}\ x = e\ \mathbf{in}\ e' : t}$$

In TCG, the quantified variables $\tilde{\alpha}$ are computed as the inner variables I of the subproof for the first premise. Soundness of this procedure of obvious, as the variables in Γ are certainly not in I, thus never among the quantified variables. For completeness, we have to argue that any variable $v \in \mathbf{FV}(s) \setminus I$ is in $\mathbf{FV}(\Gamma)$ as well. We are now going to establish

$$I = \mathbf{FV}(s) \setminus \mathbf{FV}(\Gamma) \tag{2.4.9}$$

by a proof-theoretical argument, paralleling the proof-theoretic motivation for the notion of inner variables. Suppose that the first premise is located at position p in proof P. Oberserve that when the premise is inserted to P, Γ is "outside" of the suproof at p, while s is "inside", hence (2.4.9) is satisfied. This distinction between $t\{p\}$ and $t\backslash_p$ motivates the following definition.

2.4.15 DEFINITION. Let t be a tree of judgements and $p \in dom(t)$. Define the *outer variables of the subtree* $t\{p\}$ by

$$\mathbf{OV}(t,p) := \mathbf{FV}(t\{p\}) \cap \mathbf{FV}(t\backslash_p)$$

The outer variables are extended to subproofs by $\mathbf{OV}(\langle t, f, F\rangle\{p\}) := \mathbf{OV}(t,p)$.

The outer variables of subproof $P\{p\}$ are those variables from $P\{p\}$ that cannot be inner variables of $P\{p\}$ because they have already been observed outside of that subproof. We are going to establish that a variable cannot become "outer" to p by coincidence, but only because it gets unified with some variable that is already outer to p. For motivation, note that in the let-example, when the first premise is inserted we have

$$\mathbf{FV}(\Gamma) = \mathbf{OV}(P,p) \tag{2.4.10}$$

Furthermore, a characterization of the outer variables yields a characterization of the inner variables by using $A \setminus B = A \setminus (A \cap B)$:

$$\begin{aligned} \mathbf{IV}(P,p) &= \mathbf{FV}(P\{p\}) \setminus \mathbf{FV}(P\backslash_p) \\ &= \mathbf{FV}(P\{p\}) \setminus (\mathbf{FV}(P\{p\}) \cap \mathbf{FV}(P\backslash_p)) \\ &= \mathbf{FV}(P\{p\}) \setminus \mathbf{OV}(P,p) \end{aligned} \tag{2.4.11}$$

2.4.16 REMARK. Although the outer variables are in this sense complementary to the inner variables (Definition 2.3.1), the already established results do not carry over. The inner variables of a subproof could be kept constant (Lemma 2.4.2) by restricting the applicable substitutions to those *compatible* with P (Definition 2.3.9). That restriction, in turn, is sensible only because the inner variables of complete subproofs need not be touched again. No such restriction is possible for outer variables, as they necessarily change during resolution of goals, otherwise the characterization of polymorphic let would not be possible.

The most general unifiers (Definition 2.1.10) must be restricted to obtain the result. The restriction is obeyed by standard unification algorithms for first-order terms [BS01] and DAGs, and higher-order terms [Hue75, Nip93], as these algorithms only introduce fresh variables to the unification problem, if at all. These new variables can be chosen arbitrarily, in particular they can be chosen such that they do not come from a given set U.

2.4.17 Definition. Let U be a set of *used* variables and E a set of equations. The unifier $\sigma := \mathit{mgu}(E)$ *respects* U if $\mathbf{FV}(\mathit{range}\,\sigma) \subseteq \mathbf{FV}(E) \cup \complement U$.

By Remark 2.1.11, we have $\mathit{dom}\,\sigma \subseteq \mathbf{FV}(E)$ since σ is a most general unifier.

2.4.18 Lemma. *Assume P is a proof and unification respects $\mathbf{FV}(P)$ and*

$$P \xrightarrow[R,q]{\mathit{Resolve}} P'$$

with unifier τ. Then for $p \in \mathit{dom}(P)$

$$\mathbf{OV}(P', p) \subseteq \mathbf{FV}(\tau(\mathbf{OV}(P, p))) \tag{2.4.12}$$

Proof. Both of the proof cases below conclude using the following equation (by (2.1.7)):

$$\begin{aligned}\mathbf{FV}(\tau(\mathbf{OV}(t,p))) &= \mathbf{FV}(\tau|_{\mathbf{OV}(t,p)}(\mathbf{OV}(t,p))) \\ &= \mathbf{OV}(t,p) \cap \complement\, \mathit{dom}\,\tau|_{\mathbf{OV}(t,p)} \cup \mathit{range}(\tau|_{\mathbf{OV}(t,p)})\end{aligned} \tag{2.4.13}$$

$\underline{p \le q}$ Let $P = \langle t, f, F \rangle$. Since unification respects $\mathbf{FV}(P)$ and τ is a most general unifier

$$\mathit{dom}\,\tau \subseteq \mathbf{FV}(t[q]) \subseteq \mathbf{FV}(t\{q\}) \subseteq \mathbf{FV}(t\{p\}) \tag{2.4.14}$$

$$\mathit{range}\,\tau \subseteq \mathbf{FV}(t[q]) \cup \complement\mathbf{FV}(t) \subseteq \mathbf{FV}(t\{p\}) \cup \complement\mathbf{FV}(t) \tag{2.4.15}$$

The following elementary computation yields the result. The underlined terms indicate the use of (2.4.13).

$$\begin{aligned}
\mathbf{OV}(t\tau, p) &= \mathbf{FV}(t\{p\}\tau) \cap \mathbf{FV}(t\backslash_p\tau) \\
&= \mathbf{FV}(t\{p\}\tau) \cap \mathbf{FV}(t\backslash_p\tau|_{\mathbf{FV}(t\{q\}) \cap \mathbf{FV}(t\backslash_p)}) && \text{by (2.4.14), (2.1.4)} \\
&= \mathbf{FV}(t\{p\}\tau) \cap \mathbf{FV}(t\backslash_p\tau|_{\mathbf{OV}(t,p)}) \\
&= \big(\mathbf{FV}(t\{p\}) \cap \complement\, \mathit{dom}\, \tau \cup \mathit{range}\, \tau|_{\mathbf{FV}(t\{p\})}\big) \\
&\cap \big(\mathbf{FV}(t\backslash_p) \cap \complement\, \mathit{dom}\, \tau|_{\mathbf{OV}(t,p)} \cup \mathit{range}\, \tau|_{\mathbf{OV}(t,p)}\big) \\
&= \underline{\mathbf{FV}(t\{p\})} \cap \complement\, \mathit{dom}\, \tau \cap \underline{\mathbf{FV}(t\backslash_p)} \cap \underline{\complement\, \mathit{dom}\, \tau|_{\mathbf{OV}(t,p)}} \\
&\cup \mathbf{FV}(t\{p\}) \cap \complement\, \mathit{dom}\, \tau \cap \underline{\mathit{range}\, \tau|_{\mathbf{OV}(t,p)}} \\
&\cup \underbrace{\mathit{range}\, \tau|_{\mathbf{FV}(t\{p\})}}_{\text{by (2.4.15)} \subseteq \mathbf{FV}(t\{p\}) \cup \complement \mathbf{FV}(t)} \cap \underline{\mathbf{FV}(t\backslash_p)} \cap \underline{\complement\, \mathit{dom}\, \tau|_{\mathbf{OV}(t,p)}} \\
&\cup \mathit{range}\, \tau|_{\mathbf{FV}(t\{p\})} \cap \underline{\mathit{range}\, \tau|_{\mathbf{OV}(t,p)}} \\
\text{by (2.4.13)} &\subseteq \mathbf{FV}(\tau(\mathbf{OV}(t,p)))
\end{aligned}$$

$\underline{p \not\leq q}$ Then, because $q \in \text{goals}(P)$, also $q \not\leq p$, that is $t\{p\}$ and $t\{q\}$ are disjoint subtrees of t. By analogous reasoning as before, we have

$$\begin{aligned}
\mathbf{OV}(t\tau, p) &= \mathbf{FV}(t\{p\}\tau) \cap \mathbf{FV}(t\backslash_p\tau) \\
&= \mathbf{FV}(t\{p\}\tau|_{\mathbf{OV}(t,p)}) \cap \mathbf{FV}(t\backslash_p\tau) \\
&= \big(\mathbf{FV}(t\{p\}) \cap \complement\, \mathit{dom}\, \tau|_{\mathbf{OV}(t,p)} \cup \mathit{range}\, \tau|_{\mathbf{OV}(t,p)}\big) \\
&\cap \big(\mathbf{FV}(t\backslash_p) \cap \complement\, \mathit{dom}\, \tau \cup \mathit{range}\, \tau\big) \\
&= \underline{\mathbf{FV}(t\{p\})} \cap \underline{\complement\, \mathit{dom}\, \tau|_{\mathbf{OV}(t,p)}} \cap \underline{\mathbf{FV}(t\backslash_p)} \cap \complement\, \mathit{dom}\, \tau \\
&\cup \underline{\mathbf{FV}(t\{p\})} \cap \underline{\complement\, \mathit{dom}\, \tau|_{\mathbf{OV}(t,p)}} \cap \underbrace{\mathit{range}\, \tau}_{\subseteq \mathbf{FV}(t[q]) \subseteq \mathbf{FV}(t\backslash_p)} \\
&\cup \underline{\mathit{range}\, \tau|_{\mathbf{OV}(t,p)}} \cap \mathbf{FV}(t\backslash_p) \cap \complement\, \mathit{dom}\, \tau \\
&\cup \underline{\mathit{range}\, \tau|_{\mathbf{OV}(t,p)}} \cap \mathit{range}\, \tau \\
\text{by (2.4.13)} &\subseteq \mathbf{FV}(\tau(\mathbf{OV}(t,p)))
\end{aligned}$$

●

2.4.19 Theorem. *Let $P = \langle t, f, F \rangle$ be a proof, $p \in \mathit{dom}(t) \setminus \mathit{dom}(f)$ and $P \longrightarrow^* P'$ such that unification respects the free variables of all intermediate proofs. Let σ be the composition of all substitutions applied in the derivation.*

$$\mathbf{OV}(P', p) \subseteq \mathbf{FV}(\sigma(\mathbf{OV}(P, p))) \tag{2.4.16}$$

Proof. Proceed by induction on the number of derivation steps. The claim is obvious for zero steps, where $P' = P$ and $\sigma = \varnothing$. In the induction step

$$P \longrightarrow^* \hat{P} \longrightarrow P'$$

we have two substitutions $\hat{\sigma}$ and σ' with $\sigma = \sigma' \circ \hat{\sigma}$. We have

$$\mathbf{OV}(P', p) \subseteq \mathbf{FV}(\sigma'(\mathbf{OV}(\hat{P}, p)))$$

by case distinction on the applied derivation step. For resolution use Lemmata 2.4.18. For grafting $P \leftharpoonup P'$, use the same Lemma twice, once for the renaming ρ within P' and next for the unifier σ on P and P'. For the addition of a goal, observe that subproof extraction does not modify the outer variables, because Definition 2.2.11 quantifies over the inner variables of subproofs before extracting them.

The induction hypothesis (2.4.16) yields

$$\mathbf{OV}(\hat{P}, p) \subseteq \mathbf{FV}(\hat{\sigma}(\mathbf{OV}(P, p))$$

such that finally

$$\begin{aligned}\mathbf{OV}(P', p) &\subseteq \mathbf{FV}(\sigma'(\mathbf{OV}(\hat{P}, p)))\\ &\subseteq \mathbf{FV}(\sigma'(\mathbf{FV}(\hat{\sigma}(\mathbf{OV}(P, p)))))\\ &= \mathbf{FV}(\sigma'(\hat{\sigma}(\mathbf{OV}(P, p))))\\ &= \mathbf{FV}((\sigma' \circ \hat{\sigma})\mathbf{OV}(P, p)) = \mathbf{FV}(\sigma, \mathbf{OV}(P, p))\end{aligned} \qquad \bullet$$

2.4.20 Remark. Theorem 2.4.19 with (2.4.10), (2.4.11) establishes the completeness of the let-rule. Suppose P is a proof, to which the rule's first premise has just been added at p

$$P[p] = \Gamma \vdash s$$

and $P \longrightarrow^* P'$, where $P'\{p\}$ is complete. Let σ be again the composition of all substitutions applied. This entails

$$P'[p] = \Gamma\sigma \vdash s\sigma$$

At the point where the inner variables are determined, we thus have

$$\begin{aligned}\mathbf{IV}(P', p) &\overset{\text{def}}{=} \mathbf{FV}(P'\{p\}) \setminus \mathbf{OV}(P', p)\\ &\supseteq \mathbf{FV}(P'[p]) \setminus \mathbf{FV}(\sigma(\mathbf{OV}(P, p)))\\ (2.4.10) &\supseteq \mathbf{FV}(s\sigma) \setminus \mathbf{FV}(\Gamma\sigma)\end{aligned}$$

2.5 Extensions

The formalism presented in the preceding sections exhibits the central integrity constraints of Tcg proofs and their maintenance through derivation steps. The main technical point is contained in Theorem 2.4.7 on application of substitutions to proofs, the derivation steps in Section 2.4 are well-defined by straightforward corollaries. However, the formalism does not yet establish a convenient working ground for the applications in Chapter 4: Although, for example, the prototypical language MiniML [CDDK86] can be modeled (see Section 4.1), several features found in practical type systems would require encodings and "programming" in Tcg, rather than declaratively specifying the typing rules.

This section therefore adds straightforward extensions to the formalism, which do not alter the structure of the existing definitions and proofs, yet would introduce mere technical complications that are not motivated by an additional gain of insights.

2.5.1 Rule Expressions

Type systems often include a notion of renaming of identifiers. For instance, when a module M defining an identifier x is included, the qualified name $M.x$ accesses the identifier x within M. A different form of import makes x an alias for $M.x$. This notion can be expressed succinctly in the TCG rule:

$$\forall(x,t)\frac{\vdash M.x : t}{x : t}(\text{unqualify–}M)$$

However, if a name $M.x$ is noted in the context, the rule would be applied at every reference to x, which is clearly inefficient. It would be desirable to add $x : t$ to the context for each $M.x : t$ as soon as the module M is imported. This can be accomplished by forward application with the rule (unqualify–M) if we allow rule expressions (Definition 2.2.1) to reference the context of the current goal:

$$RE ::= \cdots \mid \textsc{Env}$$

Technically, the subproof environment Φ_t^{pi} (Definition 2.3.3) needs to be augmented with the context of the judgement $t[p]$. Then the evaluation of rule expressions (Definition 2.2.11) extends to the new case straightforwardly. The proofs of the lemmata in Section 2.4.1 go through because no new dependencies on inner variables (Lemma 2.4.2) are introduced.

Subproof extractions with quantification models the soundness proofs and type inference for polymorphic let (Section 2.4.1.2, 4.1.3.3). If more than one value is to be defined simultaneously, then a single subproof extraction is not sufficient: We need one extraction per value, and these occur at several points within a subtree (Section 4.1.3.6). We therefore extend extraction with a second parameter q, which designates a path within the subtree at premise i:[6]

$$RE ::= \cdots \mid \mathcal{R}^{\forall}(i,q)$$

The subproof environment Φ_t^{pi} (Definition 2.3.3) has to be extended to contain all the proof nodes piq, then evaluation of rule expression (Definition 2.2.11) can handle the path q. Quantification, however, still occurs over the inner variables of the subproof $P\{pi\}$, not over the inner variables of $P\{piq\}$. The sequence of lemmata in Section 2.4.1 goes through, the proofs never refer to the exact shape of the extracted rule's conclusion. Also the discussion in Section 2.4.1.2 remains valid, because with the reconstructed derivation at $P\{pi\}$, we have implicitly reconstructed a derivation at $P\{piq\}$.

It is sometimes desirable to extract a subproof as a rule (Definition 2.2.11) without quantifying over its inner variables. For instance, the type check for patterns (Section 4.1.4) introduces fresh type variables for the pattern variables, and these assigned typings can be obtained by subproof extraction, but the fresh variables must not be polymorphic. The rule expressions are extended by non-quantified extraction, which includes an additional path q as in the previous extension.

$$RE ::= \cdots \mid \mathcal{R}^{\not\forall}(i,q)$$

[6]The TCG input language replaces q with more readable named *exports* (Section 3.1.3).

The interaction of non-quantified extraction with the lemmata in Section 2.4.1 is more subtle: As soon as the expression $\mathcal{R}^{\not\forall}$ is executed in an *add goal* derivation step (Section 2.4.3), the inner variables of $P\{pi\}$ do occur outside of $P\{pi\}$, hence cease to be inner variables. This step only lessens the requirement imposed by Condition 3(c)iii of Definition 2.3.3, but subproof extraction in 3(c)i will not be able to quantify over the previously inner variables. Rule expression $\mathcal{R}^{\not\forall}$ therefore must not extract a subproof that has been extracted with $\mathcal{R}^{\forall}$ before.

Thus there are two consequences of the new form of subproof extraction: First, as soon as it has been performed, a subsequent extraction with $\mathcal{R}^{\forall}$ on the same subproof is equivalent to using $\mathcal{R}^{\not\forall}$ again. Second, if $\mathcal{R}^{\forall}$ has been performed before $\mathcal{R}^{\not\forall}$, fixed variables of the proof may occur in unsolved judgments, which cannot happen with $\mathcal{R}^{\forall}$ alone, because the fixed variables become bound variables in the extracted rule. Hence, the implementation (Chapter 3) must explicitly tag fixed variables.

Summarizing, the new operation $\mathcal{R}^{\not\forall}$ introduces a dependency on the order of execution relative to $\mathcal{R}^{\forall}$. That interference occurs only when the same subproof is extracted with both operations at the same rule application (if one extraction occurs in the subproof extracted by the other reference, the order is again irrelevant due to the definitions of **IV** and **OV**). This situation, however, should rarely be encountered in practice, since a judgement is meant to model either polymorphism or monomorphism, but not both.

Both forms $\mathcal{R}^{\forall}$ and $\mathcal{R}^{\not\forall}$ can also be modified to *discharge* the deferred, hence non-resolved goals of the extracted subtree (Definition 2.3.3). Let L be those leaves. The variants

$$\mathcal{R}^{\forall}_{\textsc{Discharge}}(i,q) \qquad \mathcal{R}^{\not\forall}_{\textsc{Discharge}}(i,q)$$

in addition to extraction modify the proof $\langle t, f, F\rangle$ into $\langle t, f', F\rangle$ where

$$f' = f|_{\complement L} \cup \{l \mapsto \textsc{Discharged} \mid l \in L\}\ .$$

The range of the annotation function of Definition 2.3.3 must be changed accordingly. The new tag Discharged is treated like Deferred in the proofs of the preceding sections, the stated properties continue to hold.

2.5.2 Lists

Most realistic programming languages abound in repetition constructs: Rather than binding just one variable in `let`, they allow several to be defined simultaneously (and recursively); rather than having one (tuple) argument for a function, they allow several arguments. From the theoretical point of view, this iteration does not introduce complications, yet practical type checkers have to cope with it. For that purpose, we introduce *lists* with special constructors :: (*cons*, infix) and `[]` (*nil*) in the input language. The notation is complemented by the following mechanisms, which recognize the special constructors.

In checking a sequence of list elements, one often needs to perform the same check on each of them. The obvious way to check a predicate $p(e)$ for each element e of a list l is two use a new predicate $\bar{p}$ and two rules

$$\frac{p(e) \quad \bar{p}(e')}{\bar{p}(e::e')}\ (\text{step}) \qquad \frac{}{\bar{p}(\texttt{[]})}\ (\text{base}) \tag{2.5.1}$$

Writing these pairs of rules is a purely mechanical effort. Tcg supports iteration over lists by *iteration premises*. Simply adding an *ellipsis* ... to the premise to be solved for each element of $\bar{e}$ generates just the above rules for a new constructor $\bar{p}$ and replaces the original premise with $\bar{p}(\bar{e})$.

$$\frac{\cdots \quad p(\bar{e})\ldots \quad \cdots}{\cdots}$$

If e is not a variable, but a constructed term with several variables, then all variables are treated as lists, and they are traversed simultaneously. For instance, writing

$$\frac{\cdots \quad p(\bar{e},\bar{t})\ldots \quad \cdots}{\cdots}$$

checks $p(e_i, t_i)$ for each pair of elements $e_i \in (e_1 \mathinner{..} e_n)$ and $t_i \in (t_1 \mathinner{..} t_n)$. If $\bar{t}$ is a variable and $\bar{e}$ is a ground term to be checked, then this *simple iteration* yields $\bar{t}$ instantiated with the results of checking $\bar{e}$.

If not all variables in the premise are list variables, the *iteration variables* can be specified explicitly; the remaining variables in $\bar{e}$ remain fixed throughout the iteration.

$$\frac{\cdots \quad p(e)\ldots[x_1 \mathinner{..} x_n] \quad \cdots}{\cdots}$$

Finally, both subproof extraction and exports integrate very well with iteration constructs. Take for example the declaration sequence $ds = \big(x_1 = e_1 \mathinner{..} x_n = e_n\big)$, where every e_i may refer to each e_j with $j < i$ as a polymorphic value. This requires that the outcome of each check is entered to the context for iteration, which can be accomplished by the following rule; the reference to 1 is explained by the internal realization via two rules (2.5.1):

$$\frac{\cdots \quad ds\ldots + \mathcal{R}^{\forall}(1) \quad \cdots}{\cdots}$$

The premises in the input language thus take the following general form (see also Section 3.1.3, $\cdot^?$ denotes optional elements):

$$\mathit{branch}^? \; \mathit{term} \; (\textbf{export } \mathit{id})^? \underbrace{\Big(\ldots \mathit{cmod}'^? \big(\,[x_1 \mathinner{..} x_n]\,\big)^?\Big)^?}_{\text{specification of iteration}} \mathit{cmod}^?$$

Without iteration, the meaning is clear from the definition of proofs and rules. If an ellipsis is present, then *term* is the premise to be iterated, either by simple iteration or by iteration over $x_1 \mathinner{..} x_n$. The outer *cmod* is attached to the premise directly, while the inner *cmod′* is attached to the second premise in the iteration step (2.5.1). By using extraction on premise 1, therefore, the later steps of the iteration may refer to the results of the earlier steps.

Chapter 3

Implementation

This chapter describes an interpreter `tcg` for the calculus of Chapter 2. The main objective is to provide a platform for the applications in Chapter 4, and to explore the viability of implementing Tcg efficiently. The overall structure of the interpretation process is outlined in Figure 3.1. The input fragments $f_1 \mathinner{..} f_n$ (Section 3.1) containing the formalization of a type system are gathered by a translator, which creates a one-to-one internal representation of the contained rules (Section 3.1.3), the initial context (Definition 2.3.5), and a parser for the syntax of the formalized language (Section 3.1.2; Figure 1.2). The abstract syntax tree of the parsed program together with the initial context forms the initial goal. The interpreter resolves this goal and produces a set of complete result proofs.

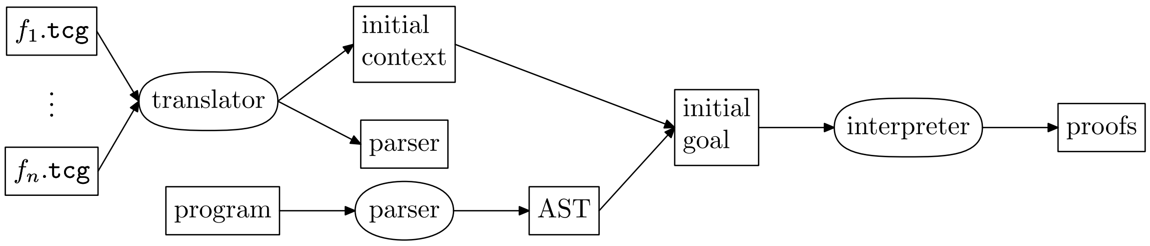


Figure 3.1: Structure of the Interpretation `tcg`

Section 3.1 specifies the Tcg input language. An operational semantics of the language constructs is given informally relative to the proof structures and derivation steps from Chapter 2. Section 3.2 briefly outlines the translation process necessary to obtain the initial goal. Section 3.3 contains the interpreter for Tcg.

3.1 Tcg Language

Let us call the pair of syntax and type system to be formalized together the *target language*. The basic structure of the Tcg input aims at accumulating that formalization as a set of atomic and independent constituents. The input consists of a set of *fragments*, each of which is contained in a single input file. Each fragment in turn is a set of the following basic elements, which will be described in the remainder of this section.

Token, Syntax, Precedence capture the input syntax of the target language. According to Section 1.1, TCG includes a LEX/YACC-based parser generator to facilitate experimental developments.

Rule A typing rule (Definition 2.2.1).

Environment The initial context, specified by references to defined rules.

Defer, Proof Grammar These instructions modify the search strategy for a proof. They are introduced in Section 3.1.4.

External Formatting of internal data- and term structures for output.

Use Inclusion of another fragment.

Each of these elements, except for rules, can be preceded by an optional *label*, which is used by TCGDOC (Section 3.6) to establish references. Rules are labeled with their names by default. The syntax of an input file is thus:

$$
\begin{array}{rcl}
\textit{fragment} & ::= & (\ (\ \textit{label}^{?}\ \textit{labeled\_element}\ )\mid \textit{rule})^{+} \\
\textit{labeled\_element} & ::= & \textit{tokens} \mid \textit{syntax} \mid \textit{precdecl} \mid \textit{use} \mid \textit{extern} \\
 & \mid & \textit{defer} \mid \textit{proofstructure} \mid \textit{env} \\
\textit{label} & ::= & \texttt{'['}\ \texttt{'label'}\ \textit{id}\ \texttt{']'} \\
\textit{use} & ::= & \texttt{'use'}\ \langle\ \textit{file name}\ \rangle
\end{array}
$$

The **use** f element can be explained directly: f is an arbitrary string that continues to the end of the input line. All period characters in f are replaced by the directory separator for filenames, then a suffix `.tcg` is appended. The resulting file is searched for in the include path given on the command line of the TCG call. The first found file is parsed and the elements from the resulting fragment are added to the current one.

3.1.1 Terms

The basic format of terms in the first production below is standard: It allows identifiers, string literals and application of function symbols. The *semantic value* notation `$`i is allowed only in the right-hand side of grammar productions (Section 3.1.2). The second line below introduces *opaque values*, or *opaques* for short. These enclose the semantic values of tokens matched in the input language, and preserve them unchanged through proof construction to the output phase. An opaque value $c[a]$ has a *class* c and an *argument* a. The class captures the token category matched, the argument is the actual semantic value as a string. The argument may also be an identifier (second alternative in the second row), in which case the term is an *opaque pattern*, and the identifier must be a variable in the enclosing rule. It can unify only with opaques that have the same class as the opaque pattern. Note that the variable will be bound to the entire opaque value, not to its argument.

$$
\begin{array}{rcl}
\textit{term} & ::= & \textbf{ID} \mid \textbf{STRING} \mid \texttt{'\$'}\ [0\text{-}9]^{+} \mid \texttt{'('}\ \textit{term}\ \texttt{')'} \mid \textbf{ID}\ \texttt{'('}\ \textit{term}^{+}\ \texttt{')'} \\
 & \mid & \textbf{ID}\ \texttt{'['}\ (\ \textbf{STRING} \mid \textbf{SEMVAL}\ )\ \texttt{']'} \mid \textbf{ID}\ \texttt{'['}\ \textbf{ID}\ \texttt{']'} \\
 & \mid & \textit{fmt} \mid \textit{special}
\end{array}
$$

The last line of the above production allows two classes of terms, the *format terms* and *special* terms. The purpose of the special terms, as defined next, is a simplified infix notation for frequently occurring symbols in type systems. The grammar as shown is slightly inaccurate: The infix symbols may be followed immediately, without whitespace, by an identifier, which becomes part of the infix symbol. For instance, `<=c` is a variant of `<=` used for "a subtype by conversion" in Chapter 4.

$$\begin{array}{rcl} \textit{special} & ::= & \texttt{'[]'} \mid \textit{term}\ \texttt{'::'}\ \textit{term} \\ & \mid & \textit{term}\ \texttt{':'}\ \textit{term} \\ & \mid & \textit{term}\ \texttt{'<='}\ \textit{term} \mid \textit{term}\ \texttt{'='}\ \textit{term} \mid \ \textit{term}\ \texttt{'=>'}\ \textit{term} \end{array}$$

The format terms are used to generate sequential textual output from internal terms.

$$\begin{array}{rcl} \textit{fmt} & ::= & \texttt{'['}\ \textit{term}^{+}\ \texttt{']'} \\ & \mid & \texttt{'[|'}\ \textit{term}\ \texttt{'|'}\ \textit{term}^{+}\ \texttt{']'} \mid \texttt{'@.'} \mid \texttt{'@\textbackslash n'}. \\ & \mid & \texttt{'@ARG'}.\ \texttt{'('}\ \textit{term}\ \texttt{')'} \mid \texttt{'@}\textit{CLS}\texttt{'}\ \texttt{'('}\ \textit{term}\ \texttt{')'} \\ & \mid & \texttt{'@}\textit{ID}\texttt{'}\ \texttt{'('}\ \textit{term}\ \texttt{')'} \ \mid \texttt{'@}\textit{STR}\texttt{'}\ \texttt{'('}\ \textit{term}\ \texttt{')'} \end{array}$$

Proceeding bottom-up, left-to-right through the term, the following output is generated for the above constructors:[1]

- `"..."` String constants (Grammar *term*) are output literally.
- `[...]` Concatenation of the elements
- `@\n` Line break, `@.` line break with flush
- `@[<`*i*`> ...]` A *box*, whose content is indented by *i* characters, relative to its first line.
- `@[|`*sep*`|` *seqs*`]` The arguments *seqs* are output in turn, with *sep* between them. Let *seq* be one of the arguments. If *seq* is a list built from the special constructors `::` (cons) and `[]` (nil) (see Section 4.1.3.5), then the list elements are output one by one, with the separator *sep* between them. The separator is again a general term, possibly containing formatting instructions.
- `@CLS(`x`)`, `@ARG(`x`)` A string representation of an opaque's class and argument.
- `@STR(`x`)` The output of x is enclosed in `"..."`, after adding escapes to make it a valid string literal for most programming languages.

Complementary to terms are the *selectors*, which specify predicates on terms. The first line below contains the recursion base: A term matches `[]` if it is a variable, the **STRING** and **NAT** if it is a constant of that value, `@CLS` c if the term is an opaque of class c. A term matches the second line $f.i.s$ if its top function symbol is f and the ith argument matches selector s. f may also be one of the special constructors defined above. Finally, `&` and `|` denote logical conjunction and disjunction of the results of the individual selectors.

$$\begin{array}{rcl} \textit{selector} & ::= & \texttt{'[]'} \mid \textbf{STRING} \mid \textbf{NAT} \mid \texttt{'@CLS'}\ \textbf{ID} \\ & \mid & \textit{selid}\ \texttt{'.'}\ \textbf{NAT}\ \texttt{'.'}\ \textit{selector} \\ & \mid & \textit{selector}\ \texttt{'\&'}\ \textit{selector} \mid \textit{selector}\ \texttt{'|'}\ \textit{selector} \\ \textit{selid} & ::= & \textbf{ID} \mid \texttt{':'} \mid \texttt{'<='} \mid \texttt{'=>'} \mid \texttt{'='} \mid \texttt{'[]'} \mid \texttt{'::'} \end{array}$$

[1]The notation follows the OCaml standard `Format` [OCa03] library.

3.1.2 Parser Generator

Tcg includes a description of the syntax of the target language. It uses Lex and Yacc [Her92] to generate the actual parser, but adapts their input format to the conception of a formalization as a *set* of elements, rather than a sequence. Token declarations can be interspersed with syntax productions and precedences are given numerically, rather than by the order in the input file.

The tokens are specified as pairs of identifiers and regular expressions. A token's semantic value is the string matched by the regular expression in the input, except for regular expressions that match a constant string, they do not have a semantic value.

$$\begin{array}{rcl}
\textit{tokens} & ::= & \texttt{'tokens'}\ (\ \mathbf{ID}\ \texttt{'='}\ \textit{regexp}\ )^*\ \texttt{'end tokens'} \\
\textit{regexp} & ::= & \mathbf{CHAR} \mid \texttt{'.'} \mid \texttt{'\textbackslash'}\ \mathbf{CHAR} \mid \texttt{'['}\ \textit{charset}\ \texttt{']'} \mid \texttt{'('}\ \textit{regexp}\ \texttt{')'} \\
 & \mid & \textit{regexp}\ \texttt{'|'}\ \textit{regexp} \\
 & \mid & \textit{regexp}\ \texttt{'?'} \mid \textit{regexp}\ \texttt{'*'} \mid \textit{regexp}\ \texttt{'+'} \\
\textit{charset} & ::= & \texttt{'\^{}'}^?\ (\ \mathbf{CHAR} \mid \mathbf{CHAR}\ \texttt{'-'}\ \mathbf{CHAR}\ )^+
\end{array}$$

The syntax productions resemble those of Yacc; however, they may contain keywords in double quotes, which are automatically added to the declared tokens. Furthermore, the final action of a production may be replaced by a term (Section 3.1.1), which will be constructed as the production's result.

$$\begin{array}{rcl}
\textit{productions} & ::= & \texttt{'syntax'}\ (\ \mathbf{ID}\ \texttt{':'}\ \textit{prodrhs}\ (\ \texttt{'|'}\ \textit{prodrhs}\ )^*\ )^+ \\
\textit{prodrhs} & ::= & (\ \mathbf{ID} \mid \mathbf{STRING}\ )^*\ \texttt{'-->'}\ \textit{term}\ \textit{prec}^? \\
 & \mid & (\ \mathbf{ID} \mid \mathbf{STRING}\ )^*\ \texttt{'\{!'}\ \mathbf{CHAR}^*\ \texttt{'!\}'}\ \textit{prec}^?
\end{array}$$

Precedence specifications can appear at the end of productions, and then have the same meaning as in Yacc: The default precedence of the rule, which is the precedence of the last token, is overridden. Precedence declarations at the top level of a fragment assign precedences to the given tokens.

$$\begin{array}{rcl}
\textit{prec} & ::= & (\ \texttt{'\%left'} \mid \texttt{'\%right'} \mid \texttt{'\%nonassoc'}\ )\ \mathbf{INT} \\
\textit{precdecl} & ::= & \textit{prec}\ (\mathbf{ID} \mid \mathbf{STRING})^+
\end{array}$$

The Lex and Yacc input files are generated from the declarations straightforwardly: The tokens are gathered from the explicit `tokens` and the productions; keywords, that is tokens with regular expressions matching a specific string, are put first in the Lex file, the order of the remaining tokens is unspecified. The Yacc file receives the precedence declarations, sorted by numeric value, and the grammar productions themselves. The action of the Yacc production is either the a literal copy of the action given between `{! !}`, or an expression that constructs the internal representation of the term on the right-hand side of `-->`. Semantic values $\texttt{\$}i$ refer to the semantic values on Yacc's stack.

3.1.3 Rules

The rules from Section 2.2 have a direct representation in the Tcg input language. The grammar also accounts for the extensions from Section 2.5. Besides the premises and conclusion, a rule also specifies its quantified variables and its *parameters*. Parameters are place holders that are replaced with concrete terms (Section 3.1.1) when the rule is referenced.

```
      rule ::= 'rule' ID ( '[' ids ']' )? ('forall' '(' ids ')')?
               conclusion premises?
       ids ::= ID (',' ID )*
conclusion ::= term
  premises ::= 'if' premise ('and' premise)*
```

Each premise of a rule has the following structure (see grammar below): A *term* defines the goal to be proven; that task can be modified by the remaining items in the *premise* production. First, the optional '[branch]' specifies that the goal is to be proven by an independent sub-search (Sections 3.4, 2.4.5, 3.3.3.1). Then, the subproof created for the goal can be *exported* (Section 2.5.1) for later reference in a rule expression. Finally, a context modifier (Section 2.2.1) is written down. The special premise `expand` is an instruction to direct the search process. Its implementation is treated in Section 3.3.4.5.

An *export* specification provides a label to be attached to the node in the proof tree: The label `*` (*propagate*) indicates that exports from the children of the node are to be propagated downward in the tree; the optional **ID** names the node as an export. The exclamation mark `!` indicates a *solved premise*: A solved premise creates no proof obligations, but is exported under the given name. It therefore serves to state facts for later reference.

The *ellipsis* ... makes the premise an *iteration premise* (Section 2.5.2). With the premise, a context modifier to be executed in each step and a list of *iteration variables* can be given optionally.

```
  premise ::= '[branch]'? term export? ellipsis? rulecmod?
            | 'expand' | '!'
   export ::= 'export' '*'? (ID '!'?)?
 ellipsis ::= '...' cmod* ( '[' ids ']' )?
 rulecmod ::= 'under' cmod*
```

The context modifiers have exactly the form from Section 2.2.1, with the extensions from Section 2.5: They insert the rules given by a rule expression or remove all rules matching a given selector.[2] The *guard* adapts the selector mechanism to exported rules. Since their conclusions will in general not be known beforehand, a guard $l \longrightarrow r$ implements conditional removal: For any rule that is added after the guard, and whose conclusion matches $l\sigma$ for some substitution σ, the removal instruction $-r\sigma$ is executed. The guard thus allows to state, for example, that for any newly inserted assumption $x : s$, any previous assumptions $x : t$ become invalid. Context modifiers will be executed in the same order in which they are found in the input file.

```
 cmod ::= '+' ruleexp |          '-' '(' selector ')' | guard
guard ::= selector '-->' selector
```

Rule expressions define new rules to be inserted into the context of some goal in the proof. They are recursively nested and are evaluated during proof construction according to Definition 2.2.11.

[2]Using selectors is merely a convenient alternative to the existentially quantified terms of Section 2.2.1.

$$
\begin{array}{rcl}
\textit{ruleexp} & ::= & \mathbf{ID}\ (\ \texttt{'['}\ \textit{term}^{+}\ \texttt{']'}\ )^{?} \\
& | & \texttt{'load'}\ \textit{term} \\
& | & \texttt{'environment'} \\
& | & \mathbf{ID}\ \texttt{'('}\ \textit{ruleexp}^{+}\ \texttt{')'} \\
& | & \texttt{'<'}\ \mathbf{INT}\ (\ \texttt{':'}\ \mathbf{ID}^{?}\ (\ \texttt{'['}\ \mathbf{ID}^{+}\ \texttt{']'}\ )^{?}\ )^{?}\ \texttt{'>'}
\end{array}
$$

The first case is a *rule reference* with optional arguments. It inserts the designated rule, with the rule's parameters replaced by the given arguments. The number of arguments must match the number of parameters, otherwise the rule expression is not well-formed. The next case loads the rules contained in the file designated by *term*. That term is output as a string using format constructors (Section 3.1.1) and taken as a file name. The case thus is just another form of inserting rule constants of Definition 2.2.1. The `'environment'` refers to the context at the rule application itself. The forward resolution $r(a_1 \mathinner{..} a_n)$ resolves the premises of the rule r against the conclusions of the results of $a_1 \mathinner{..} a_n$. The result is a rule with the conclusion of r and the premises of the a_i, in the order of occurrence (Definition 2.2.7). If $a_i = -$, then the ith premise of r will be skipped in the process.

The last case *rule extraction* `<`i`>` extracts the subproof at the ith premise of the rule; that premise must precede the premise containing the rule expression. The *parameters* following the colon specify the details of the extraction (Section 2.5.1). First, an **ID** e selects all the exports named e of the extracted subproof instead of the entire subproof. In general, when the ith premise is found at position q in the later proof, the exports labeled e will be found at some higher position q_0; if no export name is given, then $q_0 = q$ is the only extracted subproof. Furthermore, several resolution goals may be unsolved in the subproof at q_0. Therefore, for each proof node q_0 in q labeled with e, the following situation arises.

$$
\begin{array}{c}
[q_1]\Gamma_1 \vdash g_1 \quad \cdots \quad [q_n]\Gamma_n \vdash g_n \\
\vdots \\
[q_0]\Gamma \vdash g_0 \\
\vdots \\
[q]\Gamma \vdash g
\end{array}
$$

The remaining **ID**s are *options* and detail the treatment of extracted subproofs. The possible options are *quantify*, *discharge*, *inner goals* and *keep context*. Their meaning will be clarified now.

The inner variables of the subproof at g are those free variables that occur only in the subproof at g (Definition 2.3.1). Let I be the inner variables of g. If option *quantify* is set, then the result rule is quantified over the inner variables. Let $V = I$ in this case, otherwise $V = \varnothing$.

A goal among the $\Gamma_i \vdash g_i$ that does not contain inner variables gets copied unchanged to any application of the extracted rule, thus creating unnecessary copies of the same proof obligation. The *inner goals* are the goals that do contain inner variables; if the parameter *inner goals* is set, then only those goals are considered for the further processing. Let $G \subseteq \{1 \mathinner{..} n\}$ be the selected goals.

When the option *discharge* is given, all goals from G are considered solved after the rule extraction. The option's name has been adopted from natural deduction [Gen35] (under the *complete discharge convention* [TS00, Section 2.1.9]). Tcg's rule extraction mechanism in this reading captures the intermediate implication $P_1 \mathinner{..} P_n \Rightarrow Q$ [3] in the following situation as a rule. Here, the $P_1 \mathinner{..} P_n$ can be safely discharged, because they are re-introduced as proof-obligations in every use of the implication.

$$\textit{extracted rule} \approx \quad \dfrac{\dfrac{\begin{array}{c}[P_1] \mathinner{..} [P_n] \\ \vdots \\ Q\end{array}}{P_1 \mathinner{..} P_n \Rightarrow Q}\,(\Rightarrow I) \qquad P_1 \mathinner{..} P_n}{Q}\,(\Rightarrow E) \quad \approx \textit{use of extracted rule}$$

A similar reasoning underlies the Haskell type class constraints [Jon94, Sul00].

The *keep context* option specifies whether the contexts Γ_i are represented in the premises of the result rule. If it is set, the context-modifiers of these premises remove all rules and then re-insert those from the Γ_i. Otherwise, the context modifiers are empty, as in Definition 2.2.11. Then the premises will be proven in the context found at the application sites of the rules. Let c_i be the context modifier thus defined.

Rule extraction then produces the following result rule; note that the quantification $\forall V$ captures variables in the c_i and g_i.

$$\forall V \ \frac{\langle c_i \vdash g_i \rangle_{i \in G}}{g_0}$$

3.1.4 Deferred Goals and Proof Structure

The search for a proof in TCG can be guided by two means: First, goals can be excluded from processing by a predicate on their term structure; this implements the *deferred goals* of Definition 2.3.3. Second, the rules applicable at a proof node can be restricted by a *proof grammar*.

For each inference step, TCG searches a proof for its left-most open goal and then resolves that goal. However, some goals need to be deferred, either because the search space would be prohibitively large (Examples in Section 4.2.7 and 4.4.7) or because the type system requires the goals to remain as constraints. A goal is suspended from processing if it matches a *defer selector* defined below. It will be re-considered for processing in the next inference step, thus the DEFERRED annotations of Definition 2.3.3 may change; once the a deferred goal is extracted, the DEFERRED marks becomes permanent to maintain the reasoning by Condition 3(c)ii of Definition 2.3.3. When deferred goals are discharged in a subproof extraction (Section 3.1.3), the tag DISCHARGED is attached to them and excludes them from further processing. The identifier **ID** has only documentation purposes, it does not influence the proof process.

defer ::= `'defer'` **ID** `'if'` *selector*

The second means of influencing the proof search is the *proof grammar*, which is a variant of tree grammars [CDG+99] that restricts the shape of the proof tree. We cite the relevant definition for the reader's convenience.

[3]Read this implication as a short notation for $P_1 \Rightarrow \cdots \Rightarrow P_n \Rightarrow Q$.

3.1.1 Definition ([CDG+99, Section 2.1.1]). A *tree grammar* $G = \langle A, N, \mathcal{F}, R\rangle$ is composed of an *axiom set* A, a set N of *non-terminals* (with arity 0) with $A \in N$, a set $\mathcal{F}$ of *terminal symbols* (with arity) and a set R of production rules of the form $X \to \beta$ where $X \in N$ and β is a tree over $F \cap N$.

The *derivation relation* on pairs of terms is $s \to t$ if and only if $s = C[X]$ for some context C, $X \to \beta \in R$ is a rule and $t = C[\beta]$. The *language generated* by G is the trees reached from the axiom: $\mathcal{L}(G) = \{s \mid A \to^{+} s\}$ where $\to^{+}$ is the transitive closure of $\to$.

A regular tree grammar is *normalized* if all of the productions are of the form $X \to f(X_1 \mathinner{..} X_n)$ with $X, X_i \in N$.

Tcg employs the following formulation to enable partially specified proof grammars, as are necessary for *local* restrictions of the proof tree structure: Each node in the proof tree is tagged with a *non-terminal* that specifies the tree structure of the subproof. A default non-terminal $*$ is attached to proof nodes if nothing else is specified. The productions of the proof grammar are of the form

$$l \longrightarrow f(r_1 \mathinner{..} r_n)$$

where f is a terminal of the proof grammar and r_i are again non-terminals to be attached to the children of the proof node labeled l. For proof trees, the terminal f is taken to be the name of the rule applied to the proof node labeled l.

The input of Tcg reflects the latter reading more directly by the notation $f : l \longrightarrow r_1 \mathinner{..} r_n$, where f is the name of a rule. After this declaration, rule f will only be applicable at proof nodes labeled l and the goals created for f's premises are labeled $r_1 \mathinner{..} r_n$ (see Section 3.3.4.1 for the implementation).

```
proofstructure ::=  ( ID ':' pstag '-->' pstag* )*
         pstag ::=  '*' | ID
```

3.1.5 External Presentations

External representations are created from Tcg's internal terms by leftmost-outermost rewriting. The choice of the rewriting strategy is dictated by the need to base case distinctions on subterms, hence the children of a term node must still be in internal form when the term node is rewritten (see also [Pau94]). Rewriting must result in a term that contains only formatting constructors (Section 3.1.1) and string literals. That term is the output left-to-right, bottom up in a sequential form. For each desired output format, a different set of rewrite rules can be specified; the set designated by an identifier **ID**.

```
extern ::=  'external' ID ( 'forall' '(' ids ')' )? rwrule+
rwrule ::=  term '-->' term
```

3.2 Translator

The Tcg translator (Figure 3.1) converts the external representation of a target language, which may be distributed over several fragments and contain cross-references, into an internal form where all references are resolved into pointers between internal data structures (Section 3.3). Its tasks are implemented by standard techniques from compiler construction (e.g. [App98]). The assembly of rules follows named rule references and resolves the variable and parameter bindings lexically. Parameters are substituted by arguments, which have been been internalized previously in the same manner. The YACC/LEX input files are generated according to the description in Section 3.1.2.

3.3 Interpreter

An interpreter for the TCG calculus is obtained directly from the presentation in Chapter 2, which suggests an architecture with the four basic layers *terms*, *proofs*, *inference steps* and *search*; a fifth layer *GUI inspector* is added to facilitate the incremental development of the target language. The responsibilities of the layers are summarized in the following table. They will be made precise in the subsequent sections.

Layer	Responsibilities
Terms	• Substitution • Unification • Matching • Backtracking of substitutions
Proofs	• Proof structure of Section 2.3 • Structure sharing between inference steps • Structure sharing between alternative proofs
Inference	Basic steps from Section 2.4 as functions from proofs to proofs
Search	Apply inference steps in depth-first search for proofs
GUI Inspector	Display the data structures and inference steps

The operational requirements of the interpreter are dominated by backward resolution [Pau94] (or SLD-Resolution [Gal86]) of open goals during proof search. Therefore, the presentation will frequently refer to well-established principles from the implementation of Prolog systems [VR94]. However, the definitions in Chapter 2 assume that proofs are accessible as objects, whereas in Prolog the resolved goals can be discarded. This difference makes it infeasible to parallel the Prolog developments and at the same time give a one-to-one implementation of TCG's definition.

3.3.1 Terms

TCG's terms are standard first-order terms (e.g. [Gal86]). They include variables, constants and function applications. To these they add (Section 3.1.1) opaque values, opaque patterns and format terms. The representation of these does not involve particular problems. Variables are discussed in Section 3.3.1.3.

3.3.1.1 Structure Sharing

It has long been recognized [BM72] that the sharing of data structures is crucial to the efficiency of resolution provers [Rob65]: Applying the unifier in each resolution step would mean copying the term structure of the input clauses. In their landmark article [BM72], Boyer and Moore introduce a very general solution: Substitutions need never be applied at all if every term node is annotated with the substitution that would have been applied. (The pair of term and substitution has also been termed a *molecule* [Bru82, VR94]. A similar technique, *explicit substitutions*, has been re-discovered by Abadi et al. [ACCL91] to formalize the implicit renaming of the Barendregt variable convention (Convention 2.2.3).) If unification encounters a variable which is annotated with a substitution containing a binding for the variable, then it proceeds with the binding instead. New bindings are always recorded in the substitution found at the respective variable. This scheme also works in the presence of ∀-quantification: The structure of t in $s = \forall\alpha.t[\alpha]$ does not need to be copied if different uses of s are annotated with different substitutions, thus implicitly renaming α to a fresh variable for each use. (Boyer and Moore use a shared *binding environment* and introduce *indices*, i.e. small integers, that uniquely identify different uses of s. An *expression* is then a pair of term node and index. In later Prolog implementations, environments become vectors of variables, and variables in terms become offsets into the vector [FKS94, Li98]. This representation is akin to the usual environment data structures of functional languages [Jon87].)

The Boyer-Moore structure sharing scheme trades a reduction in memory consumption for an increase in run-time: Unification must visit the binding environment at each variable it encounters; on the other hand, garbage collection time and other run-time penalties for a large memory footprint will possibly be avoided. Although the implications on run-time are not resolved for either alternative [Bru82, Mel82], many later Prolog systems have adopted *structure copying* as their strategy [VR94], that is the term structure of rules is copied for each application, including the variables, which hold the substitutions as pointers.

The TCG interpreter implements structure copying as well. However, unlike Prolog rules, TCG rules are not necessarily closed, and it is essential to maximize structure sharing for non-closed rules. For example, the typing rule for λ-abstraction (Section 4.1)

$$\forall\,(x,e,s,t)\ \frac{-\,(:_1 = x);\ +\,[x : s] \vdash e : t}{\lambda x.e : (s \to t)}\ (\texttt{lambda})$$

inserts rule $x : s$ into the context. Its term structure never needs to be copied on subsequent applications, it is shared with the structure of type $s \to t$. TCG therefore distinguishes *generic terms*, which contain bound variables, from terms that the do not contain bound variables. Only the generic terms are copied. (This distinction has been adapted from the representation of types and type schemes in the OCaml compiler [OCa03].)

3.3.1.2 Lazy Paths

As an auxiliary data structure, we will need paths that indicate positions in some tree structure [Cou83]. The required operations for the data type *path* are $\cdot$ (append), $\wedge$ (longest common prefix), and $\leq$ (prefix check). These operations are readily implemented using lists, yet the run-time is unacceptable:

- If the input positions of $\wedge$ and $\leq$ have a large common prefix, that prefix is traversed without contribution to the computation. Furthermore, the prefix cannot be shared.
- In the specific application of Tcg, most of the results of $\wedge$ are never queried, because they are stored at variables that get instantiated later on.

The first problem is solved using a binary trees with shared left subtrees for common prefixes. This representation yields a logarithmic run-time in the length of input paths for both $\wedge$ and $\leq$, while $\cdot$ remains constant time. However, $\cdot$ and $\wedge$ are not functional, that is the results of two different invocations $p \cdot a$ (with paths p and element a) are not equal. The same holds for $p \wedge q$ (with paths p and q). A functional behaviour is not required, however, because $\cdot$ is used only to label new proof nodes at the time of their creation and the results of $\wedge$ operations are used only with prefix checks $\leq$, which respects the internal data structures. The second problem is addressed in the canonical way, by evaluating the operation $\wedge$ lazily (on demand).

3.3.1.3 Variables

Variables are mutable records with fields *link*, *index* and *qpath*, which represent substitutions, the variable binding and auxiliary information for subproof extraction (Section 2.2).

Substitutions are represented by destructively updating the *link* field in variables. For a binding $\{t/v\}$, v's *link* is set to the term representation of t. Access to terms dereferences the *link*s until an uninstantiated variable or a constructed term is found. (Compare to quadratic DAG unification in [BN98].)

The *index* field for bound variables consists of the *level* i and the offset j (written $\langle i, j\rangle$); for free variables, the *index* is set to FREE. Note that bound variables have a *link* field as well: In this way, unification of a goal with a rule conclusion can proceed without requiring the rule structure to be copied first, the copy can be delayed until the entire unification has been successful and the rule can be applied.

A variable's last field is its *quantification path* (or *qpath*, for short): Rule extraction with quantification (Sections 2.2.1, 4.1.3.3) needs to determine the inner variables of a subproof; for efficiency, this must be accomplished without traversing the entire term structure of the constructed proof to gather the variable occurrences [Car87, Rém92]. Therefore, the *qpath* of a variable holds the path from the root to the highest proof node below which the variable does not occur. In other words, a variable is an inner variable to the subproof at q if q is a prefix of its *qpath* field.[4]

[4] A similar technique has been employed by Rémy [Rém92] to handle generalization in polymorphic *let*: He assigns an integer *rank* to each typing judgment, as the depth of the judgement in the typing derivation. Then, he assigns a rank to a variable as the least rank of the typing judgments at which it occurs. The rank information of variables is updated during unification. In an application of the *let* rule at rank n, all the variables with rank $> n$ can be generalized.

3.3.1 Remark. The converse does not hold, that is a variable need not actually occur in the judgement indicated by its *qpath* field. If a variable occurs both in P_1 and P_2, and q_0 is the largest prefix of q_1 and q_2, then the *qpath* of v would be q_0.[5]

$$\begin{array}{c}
[q_1]P_1 \qquad [q_2]P_2 \\
\vdots \\
[q_0]P_0 \\
\vdots \\
\langle \text{root} \rangle
\end{array}$$

For bound variables, which occur only in rules and not in goals, the *qpath* field has a related meaning: A variable annotated with path $\langle i \rangle$ occurs only within the ith premise of the rule; all other variables have the empty path ε. Like in Remark 3.3.1, this annotation does not necessarily mean that they do occur in the conclusion of the rule. The paths of bound variables are called *relative paths*, because they characterize the occurrences of a variable relative to a possible application of the rule. To emphasize the difference, qpaths of free variables are also called *absolute paths.*

The relation between relative and absolute paths is depicted in Figure 3.2. During

$$\begin{tikzcd}
\boxed{\text{relative}} \arrow[loop left, "\text{forward application}"] \arrow[rr, bend left, "\text{rule application}"] & & \boxed{\text{absolute}} \arrow[loop right, "\text{rule application}"] \arrow[ll, bend left, "\text{rule extraction}"]
\end{tikzcd}$$

Figure 3.2: Interaction of Relative and Absolute Paths

rule application at proof node q, a bound variable with relative path $\langle i \rangle$ becomes a free variable with path $q \cdot \langle i \rangle$. Also during rule application, unification (see below) may modify absolute paths of variables. In the reverse direction, proof extraction (Section 2.2.1) at node q may find an inner variable v leaf node at $q \cdot p$, and with *qpath* $q \cdot p$. If that leaf becomes the ith premise of the extracted rule, then v becomes a bound variable v' with relative path $\langle i \rangle$. If, on the other hand, v occurs in several leaf nodes, or has a qpath smaller than $q \cdot p$, then v' will be annotated with relative path ε. Forward application (Section 3.3.4.3) several rules are combined to a single rule and the relative paths of the bound variables must be re-computed.

3.3.1.4 Substitutions

Application of substitutions destructively updates the *link* field of variables (Section 3.3.1.3). Structure sharing between different proofs is achieved by the standard technique of *trails*

[5]Note that Rémy's *ranks* [Rém92] (Footnote 4) would record the lower of q_1, q_2 as the rank of v, which allows v to be generalized unsoundly at that lower judgement. In the case of the *let* rule, which Rémy handles, his characterization is sufficient, because variables occur in two branches iff they also occur in the context at their highest common ancestor. In Tcg, variables may be moved through the proof by rule extraction, such that a more general mechanism is needed. As an optimization, the user may be given the option to replace paths with integers and the operations from Section 3.3.1.2 with min and $\leq$.

[Bru82, VR94, McC94], which record the modifications done during unification, such that they can be undone for other proof attempts during backtracking search. In TCG, the branch and expand mechanism (Section 3.3.3.1) requires that the trail can also be *re-done*: When the solutions from the different branches are combined, their respective substitutions must be combined as well, and search continues under the result substitution.

In the same manner, the *qpath* fields of variables must also be trailed to allow structure sharing. Therefore, a *substitution* is a pair $\langle \mathit{subst}, \mathit{proj} \rangle$, where *subst* is the trail of *link* modifications and *proj* (*projection*, for similarity with [Nip93]) is the trail of *qpath* modifications. The *proj* trail is a list of triples $\langle v, p_{\text{old}}, p_{\text{new}} \rangle$ recording the old and new value of v's *qpath* field. In the *subst* list, the old value is always the null-substitution, thus the list contains only pairs $\langle v, t_{\text{new}} \rangle$.

The basic operations on substitutions are *apply*, *undo* and $\sqcup$ (the supremum of two substitutions, Section 2.1.2; see also [Ede85, Pal90]). *Apply* and *undo* walk the substitutions and effect or revoke the recorded field modifications. The supremum $\sigma \sqcup \tau$ is computed as the most general unifier of set (Section 2.1.2; [Ede85, Proposition 4.8, Theorem 4.9]):[6]

$$\{x = x\sigma = x\tau \mid x \in \mathit{dom}(\sigma) \cup \mathit{dom}(\tau)\}$$

The chosen representation of substitutions makes the operation $\sqcup$ efficient: Walk the substitution lists of σ and τ and unify the recorded variable with the recorded binding; if the variable is in the intersection of the domains, unification will dereference any previous bindings automatically. The necessary modification of the *proj* fields are effected by unification likewise.

3.3.1.5 Unification

Unification in TCG is standard first-order unification [Rob65, BS01, BN98]: It instantiates free variables by destructively setting their *link* fields and trails these modifications in a substitution (Section 3.3.1.4), which is returned as a result. Bound variables of level 0 are treated like free variables, assuming that the intention is to instantiate the term (see below), as is the case in the inference steps (Section 3.3.4).

The only noteworthy detail are the *qpath* fields. Whenever a variable v with absolute path p is instantiated with a term $t[u_1 \mathinner{..} u_m]$, then the path q_k of variable u_k must be changed to $q_k \wedge p$ (Section 3.3.1.2), because u_k now appears at all positions where v used to occur. Following Nipkow's [Nip93] presentation of higher-order pattern unification, we call this step *projection*. If $t = u$ with path q, that is v is instantiated with another variable, then both paths p and q are changed to $p \wedge q$.

3.3.1.6 Instantiation

When a rule is applied to a goal, its premises become new goals. According to Section 3.3.1.1, the generic terms contained in the rule are not shared between applications, thus their structure is copied. At the same time, bound variables with index $\langle 0, j \rangle$ in the term structure are replaced with fresh free variables,[7] and their (instantiated) *link* term is

[6]We assume $x\rho = x$ for $x \notin \mathit{dom}(\rho)$ for any substitution ρ (Definition 2.1.1).

[7]Note the correspondence with renaming ρ in Section 2.4.2.

transferred. As mentioned before, it is essential that the copy can be postponed to the time that the rule application has been found to be successful. The separation of unification and instantiation achieves just this goal.

3.3.2 Rules and Contexts

Rules directly represent the definitions from Sections 2.2, 2.5 and 3.1.3. A rule contains *premises*, *conclusion* and *quantifier*. The quantifier contains pointers to the bound variables and is needed only during instantiation (Section 3.3.1.6) to choose a sufficient number of fresh variables, and during forward application to adjust the *qpath* fields (Section 3.3.1.3). A context modifier in a premise either adds another rule determined by a rule expression, or removes rules by a selector. A rule expression, finally, can be either a single rule, a reference to a subproof to be extracted, a forward application step, a reference to the context or a file name (Section 3.1.3).

Contexts provide operations for adding rules, removing rules based on a selector and filtering for unifiable terms. Term indexing [Gra96, RSV01] can be implemented if desired. For the example applications (Chapter 4) a simple filter for the top-level symbol has proved to be sufficient.

3.3.3 Proofs

Tcg represents constructed proofs entirely to implement the calculus of Chapter 2 as directly as possible. Proofs thus are trees of *proof nodes*, each of which contains pointers to its parent node and children to represent the tree structure. (The root node's parent pointer points to the node itself.)

Each node contains a *goal*, that is a proof obligation to be solved by backward resolution. A goal can be in two states according to Section 2.3. A *pending goal* waits for some other goals to be resolved before its context modifier can be executed. By the restriction on context modifiers (Definition 2.2.1), the dependency will be resolvable in a left-to-right processing of goals. Once a pending goal's context modifier can be executed, the goal becomes an ordinary *resolution goal*. Besides these two main proof obligations, goals can also be search instructions: *Branching goals* are resolution goals, possibly pending, that are to be solved in an independent sub-search. An *expand instruction* is resolved by expanding all preceding branch goals, that is by combining their independent solutions (Section 3.3.4.5).

A proof node has a field *solved* indicating whether its goal has been resolved already. The field has three possible values: An *unsolved* proof node still needs to be resolved; a *solved* node has been resolved, but the subproof rooted at the node does contain unsolved proof nodes; the subtree at a *complete* proof node does not contain unsolved proof nodes. Finally, a proof node has an export tag (Section 2.5) with an *export name* and an *export propagation* flag.

As in Section 2.3, the *goals of a proof* are the unsolved leaf proof nodes of the proof tree. The context will clarify whether a "goal" relates to a single proof node's proof obligation or more specifically to an open proof obligation at a leaf node.

3.3.3.1 Structure Sharing

If structure sharing for terms is important (Section 3.3.1.1) it is vital for proofs. The data structure of solved goals must not be copied at inference steps. TCG proofs therefore generalize the structure sharing of substitutions (Section 3.3.1.4) to structure sharing of proof trees. Inference steps destructively modify the pointers of a proof tree, but they keep a trail of these modifications.

This behaviour is encapsulated in the notion of *proof deltas* and *proof sets*. A proof delta is a pair $\Delta = \langle\sigma, \delta\rangle$. σ is a substitution (Section 3.3.1.4) and δ is a list of triples $\langle p, x, y\rangle$, where p is some place, that is a mutable variable, x is the old value found at p and y the new value to be installed at p.[8] We define

$$dom(\delta) = \{p \mid \exists x, y.(p, x, y) \in \delta\}$$

Three classes of places need to be modified in the proof search:

- A proof node's pointer to its children (from none to a list of children).
- A proof node's goal (from pending to resolution).
- A proof node's solved field (from unsolved to solved and complete).

Like a substitution, a proof delta $\Delta = \langle\sigma, \delta\rangle$ is *applied* by applying σ and for each $\langle p, x, y\rangle \in \delta$ in order, replacing the value x at p with value y. Note that application is partial, it is undefined if at any element of δ the expected old value x is not found. If P is a proof and Δ a proof delta, then the application of Δ to P yields a new proof P', if all elements of δ can be applied. This operation is written

$$P \xrightarrow{\Delta} P' .$$

In the other direction, a proof delta $\Delta = \langle\sigma, \delta\rangle$ can be *undone* by undoing σ and executing the elements of δ in reverse order, replacing new values with old values. Again, the operation is undefined unless the expected new values are found. The operation is written

$$P \xrightarrow{\Delta}{}^{-1} P' .$$

A defined application can be undone, that is

$$P \xrightarrow{\Delta} P' \xrightarrow{\Delta}{}^{-1} P'' \tag{3.3.1}$$

is defined if the application is defined, and then $P'' = P$.

Each inference step in Section 3.3.4 thus takes a proof and computes a proof delta to represent the new proof. The proof deltas carry tree structure themselves: At each inference step, there are in general several alternative ways to proceed with the proofs, and each is characterized by a single Δ. Writing application of deltas as arrows as above,

[8] The actual implementation keeps three lists instead of one to avoid case distinction during application. The presentation is simplified by assuming that all places can be treated alike, disregarding typing.

we have, for example:

$$\begin{tikzcd}
& & & P_0 \arrow[dll, "\Delta_{01}"'] \arrow[drr, "\Delta_{02}"] & & \\
& P_1 \arrow[dl, "\Delta_{13}"'] \arrow[dr, "\Delta_{14}"] & & & & P_2 \arrow[dl, "\Delta_{25}"'] \\
P_3 & & P_4 & & P_5 &
\end{tikzcd}$$

Such a collection of proofs is called a *proof set*. Each proof is a member of exactly one proof set, and each proof set is a tree with a designated *root proof*. The important invariant on proof sets is that at each point in time, there is some proof P in the set, such that exactly the deltas on the path from the root proof P_0 to P have been applied. P is then called *the selected proof* of the proof set. The entire structure sharing between proofs can be encapsulated in a single interface function *select_proof*(P'). Starting from the currently selected proof P of the proof set of P', it executes the necessary number of *undo* steps to make a common ancestor P'' of P and P' the selected proof. Then it applies the deltas on the path from P'' to P', thus making P' the selected proof. This operation is always defined by (3.3.1).

The tree structure of a proof set is established by a *parent* pointer each proof, where the parent of the root proof is the proof itself. Furthermore, each proof contains the proof delta from its parent to itself. This structure enables *select_proof*, as described above, to be implemented directly: From the sequence of ancestors of proofs P and P', take the greatest common prefix to find P''.

In the expand operation (Section 3.3.4.5) independently discovered proofs for independent goals must be combined into a single proof. This step can also be accomplished with structure sharing: The *merge* operation on two proof deltas is the partial function that computes the supremum of the deltas' substitutions and concatenates the modification lists:

$$\langle \sigma, \delta \rangle \sqcup \langle \sigma', \delta' \rangle := \langle \sigma \sqcup \sigma', \delta \cdot \delta' \rangle \quad \text{if } \sigma \sqcup \sigma' \neq \downarrow$$

Merge is obviously associative and we write

$$\bigsqcup \langle \Delta_1 \mathinner{..} \Delta_n \rangle := \Delta_1 \sqcup \cdots \sqcup \Delta_n$$

Furthermore, if $dom(\delta) \cap dom(\delta') = \varnothing$, then $\sqcup$ is commutative, which we indicate by

$$\Delta \dot{\sqcup} \Delta' \quad \text{and} \quad \dot{\bigsqcup} \{\Delta_1 \mathinner{..} \Delta_n\} := \bigsqcup \langle \Delta_1 \mathinner{..} \Delta_n \rangle$$

3.3.3.2 Subproof Extraction

Subproof extraction converts a subtree of a proof tree into a rule (see also Section 3.1.3). Its input consists of proof nodes p_0 at position q_0 and p at position q, such that p is an ancestor of p_0, and the proof tree rooted at p_0 may have several unsolved resolution goals.

$$\begin{array}{c}
[q_1]\Gamma_1 \vdash g_1 \quad \cdots \quad [q_n]\Gamma_n \vdash g_n \\
\vdots \\
[q_0]\Gamma \vdash g_0 \\
\vdots \\
[q]\Gamma \vdash g
\end{array}$$

The result is a rule representing the subproof at q_0. The details depend on the *extraction parameters*, as specified in Section 3.1.3. The implementation proceeds as follows.

Computation of the Inner Variables Using the *qpath* fields (Section 3.3.1.3), a prefix test against q (Section 3.3.1.2) determines which of the free variables of Γ_i, g_i and g are inner variables. The remainder of the proof need not be consulted. Let V be the inner variables.

Selection of the Premises The *inner goals* option is implemented by directly filtering the open goals by their free variables: The goal is excluded from all further steps unless it contains at least one inner variable.

Discharging Premises Depending on the application, the selected goals G are considered to be *solved* after the rule has been extracted. The parameter *discharge* is implemented by returning a proof delta that sets the proof nodes' *solved* field accordingly.

Construction of the Output Rule If *quantify* is set, then bound variables are chosen for the variables in V and those parts of the term structure of Γ_i, g_i and g_0 that contain variables from V are copied, the chosen bound variables replacing the inner variables. Let us call the results of these copies Γ_i', g_i' and g_0'. If *quantify* is not set, let these values be pointers to the original G_i, g_i and g_0 (using structure sharing from Section 3.3.3.1). The result rule is then constructed according to the *keep context* parameter. If it is set, the result rule is

$$\forall V\ \frac{\langle := (\mathrm{content}(\Gamma_i')) \vdash g_i' \rangle_{i \in G}}{g_0'}$$

otherwise, it is

$$\forall V\ \frac{\langle \varepsilon \vdash g_i' \rangle_{i \in G}}{g_0'}$$

3.3.4 Inference Steps

The presentation of the inference steps in this section proceeds top-down, starting from the *step* function at the top level. That function takes a proof P as input and seeks to extend the proof towards a complete proof. It has five possible results:

1. Failure ($\bot$): The proof cannot be completed because one of its goals is not solvable.
2. Completed ($\top$): The proof is already complete and need not be processed any further.
3. Branch $\langle P_b, P' \rangle$: Proof P_b is to be completed in an independent search and before resuming the processing of P'. Proof P_b contains a link to the branch goal in P' to which the result proofs are to be connected.
4. Proceed $\langle P_1 \mathinner{\ldotp\ldotp} P_n \rangle$: Proceed with alternative proof extensions $P_1 \mathinner{\ldotp\ldotp} P_n$.

The proofs in these results are derived from P by a single proof delta (Section 3.3.3.1). The first three cases are instructions to the calling search procedure (Section 3.4), they are returned after a simple inspection of the selected goal (Section 3.3.4.1). The last case enumerates alternative successful proof attempts for the selected goal of P; the proof deltas Δ_i are contained in corresponding the P_i (Section 3.3.3.1).

$$\begin{tikzcd} & P \arrow[dl, "\Delta_1"'] \arrow[dr, "\Delta_n"] & \\ P_1 & \cdots & P_n \end{tikzcd}$$

3.3.4.1 Goal Selection and Proof Structure

At each inference step, a single goal from the input proof is chosen for processing. Towards that end, the unsolved goals of the input proof P are scanned in order. Resolution goals and branch goals are skipped according to the *defer* declarations of the formalization (Section 3.1.4), expand goals cannot be skipped. The first non-skipped goal is selected.

Besides defer declarations, Section 3.1.4 introduces proof grammars to guide the search for proofs in TCG. That mechanism is not implemented by run-time data structures, but by modification of the input rules. When a declaration

$$r : N_0 \longrightarrow N_1 \mathinner{..} N_m$$

specifies that rule r may only be applied at proof nodes carrying non-terminal N_0, then r's conclusion c is modified to a new conclusion $N_0(c)$, that is the non-terminal is taken as a new function symbol in the conclusion term. (It is prefixed with a `@` to ensure uniqueness.) The non-terminals $N_1 \mathinner{..} N_m$ are attached to r's premises in the same manner. The special non-terminal `*` marks all proof nodes not assigned a non-terminal explicitly by the proof grammar. It does not modify the conclusion or premise at all. It must be specified also for *expand* premises that cannot carry a non-terminal.

3.3.4.2 Rule Application

Rule application is the most important and most frequent inference step in TCG. If no context modifiers (Section 2.2.1) and branches are present, this step amounts to SLD-resolution [Gal86], except that the applicable rules are taken from a local context rather than from a global set. This section contains a more detailed analysis of the interaction with structure sharing (Section 3.3.3.1) to demonstrate that context modification can be incorporated into the process with reasonable efficiency: All the cross-references between the goals introduced by proof extraction can be implemented as pointers between data structures, and the base case without modifiers can be kept simple.

Figure 3.3 shows an example in which all possible references are present. All resolution goals contain a context or context modifier, the goal to be resolved and a pointer to the goal's children. Pending goals are indicated by dashed lines. The applied rule is taken from Γ at the goal g to be resolved:

$$\frac{\mathit{cm_1} \vdash g_1 \quad \mathit{cm_2} \vdash g_2 \quad \textbf{expand} \quad \mathit{cm_3} \vdash g_3}{c} \in \Gamma$$

The context modifier cm_2 extracts the subproof at premise 1 and cm_3 extracts the subproofs at premises 1 and 2. (Both possibly name exports (Section 2.5), but this detail is immaterial for the present discussion.)

Rule application first unifies goal g with the rule's conclusion c to obtain the substitution σ. It then translates the rule's premises to goals in a left-to-right scan as sketched in Figure 3.3. During the scan, it maintains information about the preceding premises:

1. the goals created for the preceding premises
2. the currently open branches

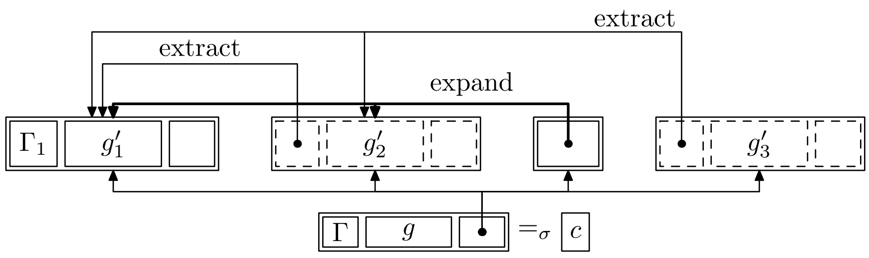


Figure 3.3: References at Rule Applications

For premise i then

1. The term g_i is instantiated (Section 3.3.1.6)
2. The context modifier cm_i is instantiated, and either
   (a) executed on Γ to obtain the context, if the modifier does not involve rule extraction (Goal 1 in the example)
   (b) translated to an internal form where the numeric references to premises are replaced by pointers to the already created proof nodes of preceding premises (Goals 2 and 4).
3. **expand** premises become **expand** goals, which reference the currently open branch goals. The list of open branch goals becomes empty.

The connection to structure sharing (Section 3.3.3.1) is seen in Figure 3.3 by the exact target point of the arrows: Each of the four goals consists of two layers (Section 3.3.3), a *proof node* and a *goal* contained in the proof node. The pointers always refer to the outer proof node, such that the later inference steps can

- Start an independent search for branch goals, and substitute the complete proof trees at the proof node during *expand.* (Section 3.3.4.5)
- Replace pending goals with resolution goals (Section 3.3.4.6) when they are selected for processing (Section 3.3.4.1). At this point, all branch goals referenced in the context modifiers must have been expanded.

3.3.4.3 Context Modification

Context modifiers (Section 2.2.1) are lists of addition or removal instructions. Their evaluation is straightforward: Starting from a context Γ_0, the execution of a context modifier $m_1 \mathinner{..} m_n$ is by iteration on $i = 1 \mathinner{..} n$:

- If $m_i = +r$ with rule expression r, then r is evaluated according to Definition 2.2.11 (see also Section 3.3.2) and the resulting rules are added to Γ_{i-1} in an unspecified order, yielding a new context Γ_i. The main challenges during evaluation are rule extraction and forward application. The extraction of subproofs as rules has been implemented in Section 3.3.3.2, forward application will be implemented in Section 3.3.4.4.
- If $m_i = -s$ with a selector s (Section 3.1.1), then Γ_i is obtained from Γ_{i-1} by deleting all rules whose conclusions match s.

3.3.2 Remark. Path-based term indexing methods [Gra96, Section 5.3, Section 6.1] can speed up not only retrieval of rules from contexts, but also their removal by context modifiers: Selectors proceed by matching (sets of) paths in terms, such that selection directly parallels the tree data structures used for indexing.

3.3.4.4 Forward Application

Forward application is requested by a rule expression (Sections 3.3.2 and 2.2.1)

$$r(re_1 \mathinner{..} re_n)$$

where r has n premises and either re_i is a rule expression or $re_i = -$. In the first case, the rules resulting from evaluating re_i are resolved against the ith premise of r according to Definition 2.2.7, in the second case the premise is retained in the result rule.

The main challenge in implementing *fwd_resolve* is the handling of bound variables. *fwd_resolve* must re-quantify the bound variables in the result rule according to Remark 2.2.9, and must re-compute their relative paths according to Section 3.3.1.3. Both goals are accomplished by the representation of bound variables, which have a *link* field for substitutions (Section 3.3.1.4). Unification can proceed as usual, yielding a unifier σ. The re-quantification is effected by choosing fresh bound variables and instantiating (Section 3.3.1.6) the input rules with these new bound variables.

Also the relative paths of the fresh bound variables are determined naturally during unification, and σ contains the necessary projections (Section 3.3.1.4). Suppose r_i is a rule obtained by evaluating re_i. Then the following tree has been constructed by forward application (it does not show the context modifiers, as they are not important for the current discussion):

$$\dfrac{\begin{array}{ccc} \dfrac{P'_{11} \mathinner{..} P'_{1m_1}}{C_1}\, r_1 & & \dfrac{P'_{n1} \mathinner{..} P'_{nm_n}}{C_n}\, r_n \\ =_\sigma & \mathinner{..} & =_\sigma \\ P_1 & & P_n \end{array}}{C} \tag{3.3.2}$$

By definition, the bound variables in the result rule must have relative qpath $\langle k\rangle$ iff they occur only in the result rule's kth premise. This outcome can be effected by unification, if we note the similarity of (3.3.2) with rule application: Suppose the kth result premise is the jth premise of the ith rule in (3.3.2). Then any (bound) variable with relative qpath $\langle i, j\rangle$ will receive relative path k in the result rule. Thus, unification handles relative paths in just the same way as it handles absolute paths: In the "small" proofs of forward application, the relative paths of bound variables play the role of absolute paths in backward proof construction.

3.3.4.5 Expand

Expanding subproofs consists of multiple simultaneous grafting operations (Section 2.4.5). The resulting proof tree and substitution is determined directly by merging the proof trees (Section 3.3.3.1). It remains to show that the operation can be carried out efficiently, in particular without the renaming of tree positions of Section 2.4.5, but retaining structure sharing (Section 3.3.3.1).

Suppose that n independent proofs $P_1 \mathinner{..} P_n$ are branched off for the judgments $J_1 \mathinner{..} J_n$ within P. After some derivation steps, these proofs are extended by $\Delta_1 \mathinner{..} \Delta_n$

$$P_i \xrightarrow{\Delta_i} P'_{i+1}$$

Now these proofs are to be grafted onto P. The merge operation (Section 3.3.3.1) efficiently joins the $\Delta_1 \mathinner{..} \Delta_n$, thereby accumulating the substitutions and extensions to the proof trees done in the $P'_1 \mathinner{..} P'_n$. Since the structures of the proof trees P'_i are pairwise disjoint, the merge can be computed in any order. However, the operation may fail if the Δ_i request contradictory instantiations of variables, such that the least upper bound of the substitutions is undefined. The desired result proof is then obtained by applying

$$P \xrightarrow{\bigsqcup\{\Delta_1 \mathinner{..} \Delta_n\}} P'$$

3.3.4.6 Pending Goals

Pending goals in Definition 2.3.3 do not occur in the constructed proof tree at all (Remark 2.3.7), they are inserted to the tree only when their context modifier can be executed. This construction simplifies the definitions, because the proof tree contains only a single form of goals. For the implementation, it is more straightforward to introduce *pending goals* directly into the data type for goals, which already allows resolution goals and branching goals. A pending goal then consists of instantiated copies of the premise to be resolved and the context modifier to be executed for the premise.

Since these copies are clearly inefficient in the most frequent cases where the context modifier is either empty or contains only elements that do not reference subtrees at all, the resolution step directly inserts premises as resolution goals, instead of creating pending goals that are converted to resolution goals later on (Section 2.4.3). Pending goals are only created in case the execution of the context modifier fails because of subproof references.

3.4 Search

Searching for proofs is essentially straightforward: Starting from a proof P_0 in its own proof set (Section 3.3.3.1), a search $S = \langle P_1 . . P_n, C_1 . . C_m\rangle$ holds the currently found partial proofs $P_1 \mathinner{..} P_n$ and the complete proofs $C_1 \mathinner{..} C_m$. By Section 3.3.3.1, all of these proofs are given by their proof delta from P_0. To extend a proof P_i, $i \in \{1 \mathinner{..} n\}$, that proof is made the selected proof, such that its pointer structure is intact for the subsequent operations. Then, one of P_i's goals is chosen according to the *defer* declarations (Section 3.3.4.1) and one inference step (Section 3.3.4) is applied to that goal. The result consists of proofs $P_{i1} \mathinner{..} P_{il}$, each with a new Δ_{ij}. The partial proof P_i in S is replaced by the $P_{i1} \mathinner{..} P_{il}$. If a proof P_i does not have a selectable goal, either because it has no open judgments, or all of its open judgments are deferred, then the proof is moved to the complete proofs as $C_{m+1} := P_i$. This process is repeated until no more partial proofs remain.

To account for a branch goal, the main search forks off a new, independent sub-search to solve this goal. The main search continues when the sub-search is finished and then grafts (Section 3.3.4.5) the found proofs onto the partial proof, replacing the branch goal. Since branching is in general recursive, the overall search data structure is a stack of sub-searches.

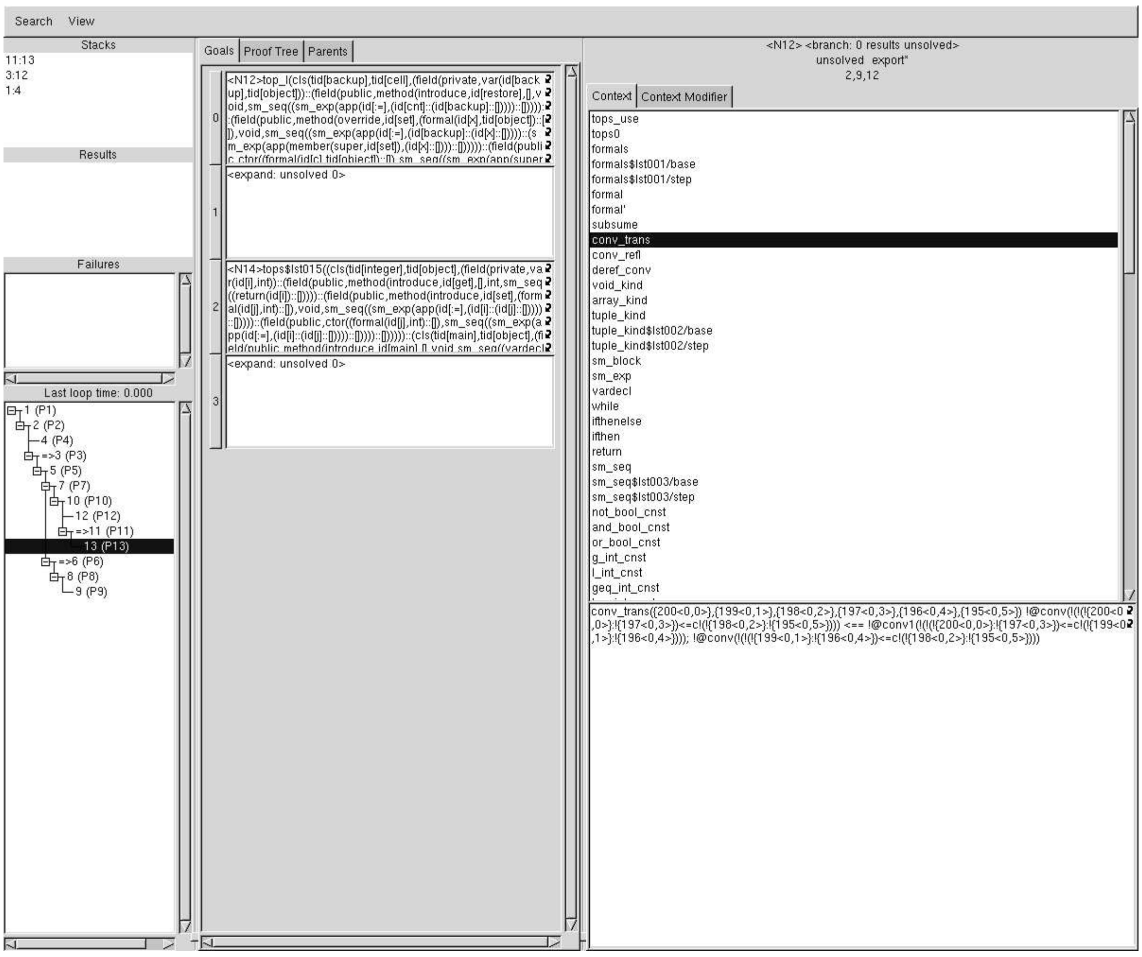
The search strategy within each sub-search is depth-first in the current implementation, although the data structures presented in the preceding sections would also support a breadth-first or mixed approach. The partial proofs of a search are treated as a stack, by selecting P_n for processing in each inference step and replacing it with the result proofs.

The shared data structures of proofs (Section 3.3.3.1) work optimally for depth-first search without branches. Then the proof deltas are accumulated for each inference step until the proof is complete. They must be undone only in case of backtracking. This procedure parallels standard Prolog implementations [VR94] where substitutions are undone upon backtracking.

3.5 GUI Inspector

The input language and formalization of TCG is designed to make the descriptions of type systems as declarative as possible. Nevertheless, the declarations may contain errors, in that the generated type checker does not behave as expected. Very often, the causes for the short-comings lie in the details of the search process: A single term constructor is misspelled, a single rule is missing from the initial context, a single forward application fails. In order to test and debug type system descriptions, it is therefore essential to make these details accessible to the user in a straightforward, convenient fashion. The GUI inspector yields a graphical method of investigating the single constructs of the data structures, ranging from the overall search structure to the context modifiers of pending goals.

Figure 3.4 shows a screenshot of the main window. It is divided horizontally into three areas for the *search*, the *proof tree* and the *context and context modifier*. As a basic principle, selection of an item in one area shows its details in the next area to the right.



```
11:13
3:12
1:4
```

```
1 (P1)
  2 (P2)
    4 (P4)
    =>3 (P3)
      5 (P5)
        7 (P7)
          10 (P10)
            12 (P12)
            =>11 (P11)
              13 (P13)
      =>6 (P6)
        8 (P8)
          9 (P9)
```

```
0 <N12>top_l(cls(tid[backup],tid[cell],(field(private,var(id[back
  up],tid[object]))::(field(public,method(introduce,id[restore],[],v
  oid,sm_seq((sm_exp(app(id[:=],(id[cnt]::(id[backup]::[]))))::[])))):
  :(field(public,method(override,id[set],(formal(id[x],tid[object])::[
  ]),void,sm_seq((sm_exp(app(id[:=],(id[backup]::(id[x]::[]))))::(s
  m_exp(app(member(super,id[set]),(id[x]::[])))::[])))))::(field(publi
  c_ctor((formal(id[c],tid[object])::[]),sm_seq((sm_exp(app(super
1 <expand: unsolved 0>
2 <N14>tops$lst015((cls(tid[integer],tid[object],(field(private,va
  r(id[i],int))::(field(public,method(introduce,id[get],[],int,sm_seq
  ((return(id[i])::[]))))::(field(public,method(introduce,id[set],(form
  al(id[j],int)::[]),void,sm_seq((sm_exp(app(id[:=],(id[i]::(id[j]::[]))))
  ::[]))))::(field(public,ctor((formal(id[j],int)::[]),sm_seq((sm_exp(a
  pp(id[:=],(id[i]::(id[j]::[]))))::[]))))::[])))))::(cls(tid[main],tid[object],(fi
  eld(public,method(introduce,id[main],[],void,sm_seq((vardecl
3 <expand: unsolved 0>
```

```
<N12> <branch: 0 results unsolved>
unsolved  export"
2,9,12
```

```
tops_use
tops0
formals
formals$lst001/base
formals$lst001/step
formal
formal'
subsume
conv_trans
conv_refl
deref_conv
void_kind
array_kind
tuple_kind
tuple_kind$lst002/base
tuple_kind$lst002/step
sm_block
sm_exp
vardecl
while
ifthenelse
ifthen
return
sm_seq
sm_seq$lst003/base
sm_seq$lst003/step
not_bool_cnst
and_bool_cnst
or_bool_cnst
g_int_cnst
l_int_cnst
geq_int_cnst
```

```
conv_trans({200<0,0>},{199<0,1>},{198<0,2>},{197<0,3>},{196<0,4>},{195<0,5>}) !@conv(!(!(!{200<0
,0>}:!{197<0,3>})<=c!(!{198<0,2>}:!{195<0,5>}))) <== !@conv1(!(!(!{200<0,0>}:!{197<0,3>})<=c!(!{199<0
,1>}:!{196<0,4>}))); !@conv(!(!(!{199<0,1>}:!{196<0,4>})<=c!(!{198<0,2>}:!{195<0,5>})))
```

Figure 3.4: Screenshot of the GUI Inspector

Search Starting from top to bottom, the first field shows the stack-of-stacks data structure of Section 3.4. The second field contains the complete proofs of the main search. The third field contains the *failures.* Whenever a selected goal in a proof cannot be resolved by any inference step, the search terminates for that proof and the proof is noted as a failure. The field in the GUI inspector provides for a further filter: If any parent goal of the failing goal either has been solved already or may be solved in the future within some proof still in the search stacks, then the failing proof is not (yet) displayed in the *failures* field. This filtering is most sensible for case distinctions by different typing rules: Since at most one rule will succeed, the user should not have to deal with the failures of the other cases. The *failures* field will contain the proofs of the case distinction as soon as *all* alternatives have been found to fail. The fourth field shows the search space as a tree, including the already processed search nodes. Alternatives are denoted by single lines, while branches are indicated by =>, and the structure of the sub-search is shown in a new tree besides that symbol.

Proof Tree The second area shows three views on the proof tree for the currently selected search node. The *goals* list contains the currently open goals, the *proof tree* exhibits the entire constructed proof tree. In these views, a single goal can be selected. Its context (or context modifier for pending goals) is displayed in the third area, and its ancestor nodes in the proof tree are displayed in the *parents* list.

Context The context display is rudimentary at the moment. It shows the names of the rules in the currently selected goal. The rule names are taken directly from the Tcg input, such that the user can identify them easily. For generated rules, such as proof extraction and forward application, the names of the original rules are maintained in the result rule's input. The extracted rules are named with the point of rule extraction. When a rule is selected from the list, its term structure is displayed in the lower part of the context area. Context modifiers are displayed in a similar manner.

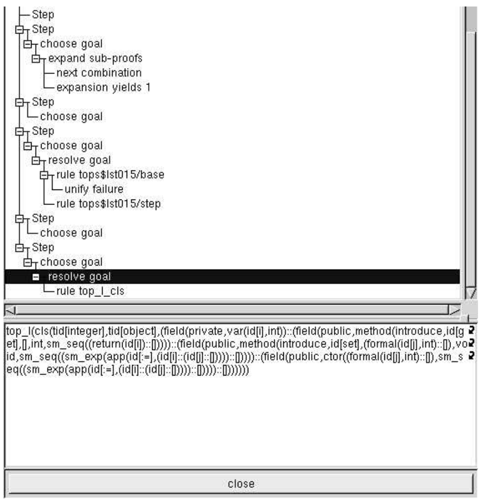
Tracker Displaying the elements of the interpreter's data structures yields a merely static picture: The display can only capture the current state at a specific point in time. The search for proofs fails, however, between these snapshots, in the applied inference steps. The *tracker* module of the implementation accounts for this circumstance by noting the key events in the inference steps and making them accessible to the user. For example, it notes goal selection, tried rules in resolution and failing unification. The amount of data saved for the single events is very small, the records consist of a few pointers to the data structures involved in the event. The events have a tree structure that corresponds tightly to the call tree of the functions implementing the inference steps. The data is discarded after a configurable number of inference steps to limit the memory consumption.

Figure 3.5 shows the GUI view on the data of the tracker. It displays the tree data structure of the events and adds the details of the attached data when one event is selected.

3.6 Documentation Generator

The syntax of the Tcg input language (Section 3.1) is designed to reflect the internal structure of the formalization (Chapter 2) directly. This choice entails that the written rules' details are intelligible to the fluent user of Tcg, although they may not appeal to the expectations of the casual reader who wants to comprehend the workings of a single type system expressed in Tcg. The documentation generator TcgDoc bridges the gap between the internal details and the conventional external presentations, by translating the rules to LaTeX output reminiscent of conventional typing rules.

The architecture of TcgDoc is geared towards extensibility in two directions: First, the output should be adaptable to different formats, ranging from visual LaTeX representations to mere markup of the input in HTML. Second, as type systems are specified as sets of individual and largely independent fragments, the representation of the typing rules should also be independent and contained in the fragments themselves (see also Section 4.2). The solution to both challenges is well-established (e.g. [Pau94]): TcgDoc generates the output through an indirection of rewriting rules, proceeding in four steps:



```
Step
Step
  choose goal
    expand sub-proofs
      next combination
      expansion yields 1
Step
  choose goal
Step
  choose goal
    resolve goal
      rule tops$lst015/base
        unify failure
      rule tops$lst015/step
Step
  choose goal
Step
  choose goal
    resolve goal
      rule top_l_cls
```

```
top_l(cls(tid[integer],tid[object],(field(private,var(id[i],int))::(field(public,method(introduce,id[g
et],[],int,sm_seq((return(id[i])::[]))))::(field(public,method(introduce,id[set],(formal(id[j],int)::[]),vo
id,sm_seq((sm_exp(app(id[:=],(id[i]::(id[j]::[]))))::[]))))::(field(public,ctor((formal(id[j],int)::[]),sm_s
eq((sm_exp(app(id[:=],(id[i]::(id[j]::[]))))::[]))))::[])))))
```

Figure 3.5: Tracking Actions

1. Read the input fragment in to an AST.
2. Gather from the fragment, and all of its `used` fragments recursively, the *external* declarations relating to the desired output format.
3. Translate the AST into a term representation.
4. Rewrite the term representation to a normal form under the accumulated external declarations.
5. Output the resulting term. This step fails if the term contains non-format constructors (Section 3.1.1).

Step 3 takes care also of the static binding of variable names in the TCG input language (see also Section 3.2) by replacing an identifier x with opaques `VAR`[x], or `PARAM`[x] if it is statically bound in a rule. Otherwise, x is tagged as `ID`[x]. Using opaque patterns, the external specifications can adapt the rendering for the different classes of identifiers. Each internal node of the AST data structure is represented directly by a term constructor in upper-case letters. Step 4 employs a leftmost-outermost strategy to allow case distinctions on child terms (see also Section 3.1.5).

The output of TCGDOC consists of one file per labeled element of the input fragment, where *labeled* refers to the `[label]` construct of Section 3.1.5. The name of the output file is the concatenation of the path and name of the input and the label. Rules are always labeled with their names, because they are most frequently referred to.

The tool `tcgdoc` takes a sequence of input fragments and processes them in order as described above. If one of the inputs is a directory, that directory is searched recursively for files with suffix `.tcg`, and these files are processed in turn. `tcgdoc` accepts a command line option `-p` *prelude*, which causes it to gather the external declarations from *prelude*`.tcg` before processing the main input files. Typically, that prelude contains the external declarations for the term constructors of AST nodes introduced in Step 3 above. The command line option `-I` adds an include directory to the search path for `use` directives.

Chapter 4

Applications

This chapter contains applications of TCG. The implemented languages are prototypical examples: Real-world programming languages tend to exhibit many particularities that have been introduced in the course of their development and their deployment. We will therefore concentrate on their main typing mechanisms, thus implementing representatives for families of languages rather than specific languages. The representatives are sufficient, however, to demonstrate that with some purely technical effort the remaining details can be provided.

TCG also constitutes a framework for comparing languages, and for exploiting commonalities in their formalization. Section 4.2 therefore establishes a set of basic language constructs that together form a library of TCG fragments to be reused in the subsequent exploration of specific typing features.

The rule names in this chapter obey the following convention: Type-writer names indicate that the rule is generated from the TCG source files using the tool TCGDOC (Section 3.6). All other rules are labeled in Roman font, probably with symbols.

4.1 Exploring Tcg

Most of the TCG formalization of the later applications is based on a few recurring patterns. These patterns also form a common ground of type systems, and they explain and motivate the definitions in Chapter 2. They are illustrated by the series of typed λ-calculi presented in this section. At the end of the sequence, we will have treated MINIML [CDDK86], which serves as a canonical object of study in the literature.

The presentation in this section deliberately repeats and summarizes parts of the input language (Section 3.1) to keep the text self-contained. The development of Section 1.2 is re-traced to exhibit the consequences of the design decisions taken there.

In the chosen family of typed λ-calculi, the difference between type-checking and type-inference consists in changing type equality checks into (syntactic [BS01]) unification constraints over types [Hin69, Mil78, Wan87, Sul00]. Since syntactic unification is available at the meta-level, we do not require type annotations in the programs.

$$\text{(cnst)}\quad \varnothing \rhd c : \sigma \quad (c : \sigma \text{ a constant}) \quad \text{(var)}\quad \Gamma, x : \sigma \rhd x : \sigma$$

$$\text{(add hyp)}\frac{\Gamma \rhd M : \sigma}{\Gamma, x : \tau \rhd M : \sigma}$$

$$(\to\text{Intro})\ \frac{\Gamma, x : \sigma \rhd M : \tau}{\Gamma \rhd (\lambda x.M) : \sigma \to \tau} \qquad (\to\text{Elim})\frac{\Gamma \rhd M : \sigma \to \tau,\ \Gamma \rhd N : \sigma}{\Gamma \rhd MN : \tau}$$

Table 4.1: Rules for Curry-style type inference [Mit90]

4.1.1 Simply-Typed Lambda Calculus

The simply-typed λ-calculus lies at the heart of almost every programming language: It embodies the basic requirements that a variable has a single type throughout its scope and that in a function application, the actual parameter's type matches that of the formal parameter. The introductory presentation by Mitchell [Mit90] gives the typing rules shown in Table 4.1: The typing judgement $\Gamma \rhd M : \sigma$ checks that M has the type σ under the assumption that the variables in M have the types assigned to them by Γ. The typing context Γ is a relation $\{x_1 : \tau_1 \mathinner{..} x_n : \tau_n\}$ between variables and types where the x_i are pairwise distinct. Rule (var) allows to retrieve the type of x in Γ. The structural rule (in the sense of [Gen35]) (add hyp) provides weakening, in that extraneous elements of Γ that are not needed in the proof of $M : \sigma$, may be discarded without changing the type of σ. The assignment of types to constants (cnst) will be treated in Section 4.1.2.

The remaining two rules check functions and their application. Rule ($\to$Intro) assigns to a term $\lambda x.M$ a function type $\sigma \to \tau$, where σ is the type assigned to the parameter x throughout the body M and τ is the result type of M. The notation $\Gamma, x : \sigma$ also includes the removal of any previous entry for x, thus effectively expressing shadowing of variables. The complementary rule ($\to$Elim) types a function application $M\,N$ by checking that M is a function and that N yields a result of M's parameter type.

Table 4.2 (Page 91) shows the TCG input formalizing the above type system. The file consists of series of top-level *elements*, each of which starts with corresponding keyword and an optional label. The label is used by TCGDOC for extracting specific elements for documentation. (Rules are labeled with their name by default.) There are five basic categories of elements: *Tokens* and *syntax* describe the input grammar and the construction of the abstract syntax tree. *Rules* state the type checking rules, which can be inserted to the context by *environment* lists. Finally, the *external* elements declare rewrite rules from the internal terms to trees of strings and other formatting instructions, which can then be output textually. The rewrite rules are split into groups by the identifier following the `external` keyword; in the example, `latex` rules are used to generate the documentation shown in this section. We now consider the single elements in more detail.

Tokens introduce lexical elements for the input grammar. They are specified by a name and a regular expression. If the regular expression is non-constant, then the matched string from the input is accessible by the token name. In the example, we have identifiers starting with a lower-case letter, followed by any alpha-numerical character or the underscore.

The input grammar is specified by syntax declarations in a YACC-like format with a distinguished start non-terminal `file`. In the example, the top-level file grammar is a

```
[label tokens]
tokens
ID=[a-z][a-zA-Z0-9_]*
end tokens

[label file] syntax
file: top_wrap EOF {! run ~save: [ (input.base^".rls",
                                    [ ("define", "save_defined") ]) ] $1 !}
top_wrap: tops  --> tops($1)
tops: top       --> $1 :: []
| top ";;" tops --> $1 :: $3
top: exp --> $1

[label exp] syntax
exp: ID                --> id[$1]
| exp exp              --> apply($1,$2)
| "\\" ID "." exp      --> lambda(id[$2],$4)
| "(" exp ")"          --> $2

rule apply
forall(f,e,s,t)
   apply(f,e) : t
if  f : fun(s,t)
and e : s

rule lambda
forall(x,e,s,t)
   lambda(x,e) : fun(s,t)
if e : t
under -( :.1.= x) + [ x : s ]

rule tops
forall(es,ts)
  tops(es)
if [branch] es : ts export* ...
environment apply,lambda,tops

external latex
forall(tops,id,x,e,f)
tops(tops)  --> @[|@\n| tops ]
id[id]      --> [ "\\mathrm{" @ARG(id) "}" ]
lambda(x,e) --> [ "\\lambda" x "." e ]
apply(f,e)  --> [ "(" f "\," e ")" ]

external latex
forall(s,t)
fun(s,t)    --> [ "(" s "\\to" t ")" ]
```

Table 4.2: Simply-Typed Lambda Calculus

sequence of expressions, separated by double semi-colons.

The right-hand sides of productions either specify an abstract syntax tree (AST) to be constructed (after `-->`) or an action to be executed (within `{! !}`). AST construction can be observed at the `tops` and `exp` non-terminals. In `tops`, the constructors a`::`b and `[]` will be transformed into simple terms `::`(a, b), `[]`; however, these *list constructors* enjoy support of several auxiliary built-in functions that will be shown in the subsequent introduction. The production `exp: ID` exhibits the construction of an *opaque* term `id[$1]`; opaque terms will be discussed further in Section 4.1.2. For this introductory example, we will not consider associativity and precedence of operators; they will be used in Section 4.2.2.

The non-terminal `file` must always execute an action to initiate the type checking process. Usually it is sufficient to apply the provided command `run` to the file's abstract syntax tree. Optionally, the results of the type checking process can be extracted and saved to files; the first argument to `run` will be discussed in Section 4.1.5.

The two type checking rules `apply` and `lambda` represent the orignal rules (→Elim) and (→Intro). Using the given `external` declarations, TcgDoc transforms the source

```
rule apply
forall(f,e,s,t)
   apply(f,e) : t
if  f : fun(s,t)
and e : s
```

```
rule lambda
forall(x,e,s,t)
   lambda(x,e) : fun(s,t)
if e : t
under -( :.1.= x) + [ x : s ]
```

into LaTeX to generate the following output. After this introductory Section 4.1, we will only show the external form.

$$\forall\,\left({\scriptstyle f,e,s,t}\right)\;\frac{\begin{array}{c}\vdash f:(s\to t)\\ \vdash e:s\end{array}}{(f\;e):t}\;(\texttt{apply})$$

$$\forall\,\left({\scriptstyle x,e,s,t}\right)\;\frac{-\,(:_1 = x);\,+\,\left[x:s\right]\vdash e:t}{\lambda x.e:(s\to t)}\;(\texttt{lambda})$$

The rule `apply` can be read directly as a Horn-clause that specifies how to prove a goal `apply(f,e):t` by backward resolution. Differing from the presentation of Table 4.1, the context Γ of the judgments is implicit in the rules and relative to the point of rule application (Section 1.2.3.2): Goals in proofs have the form $\Gamma \vdash p$, and if a rule is applied at this proof node

$$\frac{\Gamma_1 \vdash q_1 \,.\,.\, \Gamma_n \vdash q_n}{\Gamma \vdash p}$$

then the Γ_i have been derived from Γ through *context modifiers*. These context modifiers appear on the left-hand-side of the turnstile in the rules above, and after the `under` keyword in the source code. If the context modifier is empty, then the context Γ is copied unchanged.

Rule `lambda` contains an example of a context modifier. To obtain the context of the premise from that of the conclusion, the rules assigning a type to x are removed.

This is accomplished by a *selector* (Section 3.1.1) defining a predicate on terms. The expressible predicates concern paths from the root of the term: They check intermediate function symbols and descend into the given argument positions recursively. At the leaves of selectors, an atomic check for term equality and other properties can be specified. If a rule in the context satisfies the predicate, it is removed. In the example, the predicate

$$:_1 = x$$

states that the top symbol must be : and its first argument must be equal to x. Having thus removed previous bindings for identifier x, a new rule (without premises)

$$\frac{}{x : t}$$

is entered. This is an anonymous *inline rule*, as opposed to top-level, named rules. The internal treatment of both forms of rules is identical. In general, all rules have the same shape as the top-level rules, there is no restriction on the structure for recursively nested rules. As a convention, rules with empty premises will be shown without a line, as $x : t$ in the example `lambda` above.

The inserted rule $x : t$ removes the need to reproduce rule (var) in the TCG input: Proof construction proceeds by backward resolution, and to resolve a goal $\Gamma \vdash p$, all rules from Γ are tried in turn. Thus, if a goal $\Gamma \vdash x : t$ arises, the rule inserted at the nearest binding λx will be applied by resolution. The rule (add hyp) does not need a counterpart in TCG either, because the basic formalism includes weakening (Remark 2.3.8).

Finally, the documentation generator TCGDOC requires a translation of all internal term structures to *format terms* (Section 3.1.1). Format terms are special terms that contain only a few reserved constructors that request a specific sequential representation of a term.

This finishes the first exposition of the TCG input language. More constructs will be added in the subsequent sections.

4.1.2 Constants

Constants and primitive data types are added in a straightforward manner in input file `cnst.tcg`. Its first line `use lambda` causes the fragment `lambda` defined in Section 4.1.1 to be read and parsed. Its elements are then added to the current fragment as if they had been placed there literally. The language syntax is extended by new tokens

```
tokens
INT=[0-9]+
STR="[^\"]*"
end tokens
```

which are subsequently entered to the expression grammar:

```
syntax
exp:
  INT     --> int[$1]
| STR     --> str[$1]
| "false" --> bool["false"]
| "true"  --> bool["true"]
```

In these productions, the literal values are kept in *opaque terms* (or *opaques*, for short). For example, `int` is the *class* of the opaque, and then semantic value `$1` is its *argument.* Opaque terms cannot be analyzed during type inference, but the argument and class can be separated for output. Strings and booleans are treated likewise. (Note that the constant tokens `"true"` and `"false"` will not yield their string as a semantic value, hence the opaque argument has to be provided explicitly.)

The following rule `cnst_int` uses the *opaque pattern* `int`[i] to recognize integer literals for type checking: The variable i will get bound to the *entire opaque* (as oposed to its argument) during unification and matching. Thus the rule assigns type `int` to every integer literal. The same technique applies to string and boolean literals.

$$\forall\,({\scriptstyle i})\ \texttt{int}[i] : \texttt{int}\ (\texttt{int\_cnst})$$

Primitive operations are added by rules without premises, for example:

$$\text{add} : (\texttt{int} \to (\texttt{int} \to \texttt{int}))\ (\texttt{add\_cnst})$$

Conditionals likewise are straightforward: The syntax is

```
syntax
exp: "if" exp "then" exp "else" exp --> ifthenelse($2,$4,$6)
```

and their type checking rule is

$$\forall\,({\scriptstyle e1,e2,e3,t})\ \frac{\begin{array}{l}\vdash \mathit{e1} : \texttt{bool}\\ \vdash \mathit{e2} : t\\ \vdash \mathit{e3} : t\end{array}}{\textbf{if}\ \mathit{e1}\ \textbf{then}\ \mathit{e2}\ \textbf{else}\ \mathit{e3} : t}\ (\texttt{ifexp})$$

We skip the *external* declarations that have produced the above output of the rules, they contain no new language constructs.

4.1.3 Bindings

The construct `let` x `=` e `in` e' binds the value of e to the identifier x for use in e'. Its semantics can be defined as $(\lambda\, x.e')e$ (although the typings of the two expressions differ in the treatment of polymorphism (Section 4.1.3.3)).

4.1.3.1 Monomorphic Let

The typing rule for monomorphic `let` is derived from those of λ-abstraction and application: x is given a single type that must satisfy all of its uses in e.

$$\frac{\Gamma \vdash e : s \quad \Gamma, x : s \vdash e' : t}{\texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e' : t}$$

A type checker would implement the rule in three steps:

1. Type check $e : s$.
2. Assign the type s to x during a type check of $e' : t$.
3. The type of the entire expression is t.

There are two aspects in the checker that are not present in the rule: First, the *order* of the type checks is fixed, as the type of e must be known before the type of e' can be computed. Second, the type s is *transferred* from Step 1 to Step 2. Because TCG is to yield an operational type checker, the formalization must specify these two additional aspects. As the order of premises in TCG rules is fixed, the first aspect is solved. Only the transfer of the type s remains to be effected.

An obvious solution is to exploit the unification of proof search, which leads to a rule in the spirit of logic programming (the context modifier is the same as in `lambda`, Section 4.1.1):

$$\forall\,(x,e,e',s,t)\ \frac{\begin{array}{l}\vdash e : s\\ -\,(:_1 = x);\ +\,[x : s] \vdash e' : t\end{array}}{\textbf{let}\ x\ =\ e\ \textbf{in}\ e' : t}\ (\texttt{let\_prolog})$$

This formalization is sufficient for the case at hand and has been used in previous attempts to generating type checkers for ML [CDDK86, Des84]. However, it does not generalize properly:

- Polymorphism requires the variables $\tilde{\alpha} = \mathbf{FV}(s) \setminus \mathbf{FV}(\Gamma)$ to be computed. Then x is assigned not the type s, but the type scheme $\forall\tilde{\alpha}.s$ [Mil78, DM82]. Clément et al. [CDDK86] hide this computation in specialized predicates `freevars`, `setminus` and `bind`, which are implemented separately from their tool's core calculus TYPOL.
- Haskell [WB89] adds type classes to ML to treat overloaded identifiers. Technically, type classes have been considered as *constraints* on type variables [Jon94, Sul00]: The typing judgment has the form

  $$C, \Gamma \vdash e : t$$

  where $C = \{c_i(\alpha_i)\}_{i=1}^{n}$ specifies that type α_i must be in class c_i. (The generalization to constructor classes by Jones [Jon95] adds nothing to the current investigation.) Thus for Haskell, it will never be sufficient to transfer only a single type-term without transferring the associated constraints.[1]

[1]Of course, it is possible to write a type checker using backward resolution only by essentially "Typing Haskell in Prolog" (cf. [Jon99]). But this approach destroys the chance to employ meta-level mechanisms to ease the implementation.

4.1.3.2 Monomorphic Let in Tcg

TCG supports the above generalizations by the *rule extraction* mechanism, which is integrated tightly with the structure of proofs introduced in Chapter 2. Consider the proof tree for an application of the `let` rule, as it has been constructed when the second premise is to be solved:

$$\frac{\begin{array}{c}\vdots\ \text{(proof of } e : s\text{)}\\ \Gamma \vdash e : s\end{array} \qquad \Gamma' \vdash e' : t}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e' : t}$$

TCG takes the view that the *proven fact* $e : s$ is to be used within Γ'. Note the contrast with the earlier intention of transferring the type s alone: Whereas $e : s$ is a goal to be proven, and thus is a proper object of the TCG formalism, the type s depends on the specific interpretation of the binary predicate :, thus belongs to the level of the formalized type system.

Rule extraction is a primitive step that turns a subtree of the proof (or a *subproof* for short) into a rule. Briefly speaking, (see Sections 1.2.3.5, 3.1.3, 3.3.3.2)

- the subtree's root becomes the rule's conclusion and
- the subtree's unproven leaf judgments, if any, become the rule's premises.

In this setting, subproofs and rules are complementary expressions of the same proven fact at the subproof's root: Whereas the subproof represents goals that *have to be proven*, or have successfully been proven, the rule takes a more active role in *using* the already proven goals. For the operational behaviour of TCG, rules can best be understood as *knowledge* that can be used to derive further knowledge by proof construction.

Since unproven goals of the extracted subtree remain as premises, rule extraction can be applied even to incompletely constructed subproofs without destroying soundness: Whenever the extracted rule is used to resolve some goal, its premises re-enter the proof as unproven goals.[2]

Extracting the subproof is not enough, however: In the `let` rule, we will need the assumption that x, instead of e, has type s. This new reasoning step can be accomplished by *forward application* of rules. (Note that the $\forall$-quantifier below is justified by using the second rule only in the special case of a let binding.)

$$\left.\begin{array}{c}\dfrac{\langle\text{goals of subproof}\rangle}{e : s}\\ +\\ \forall(e', t)\ \dfrac{e' : t}{x : t}\end{array}\right\} \Longrightarrow \frac{\langle\text{goals of subproof}\rangle}{x : t} \tag{4.1.1}$$

Considering again rules as pieces of knowledge, forward application derives new knowledge from existing knowledge in a single step. Of course, forward application generalizes to more than one premise.

[2]Sulzmann [Sul00, Section 3.5.1] discusses whether the $\forall$-introduction rule in constraint type systems should discharge the constraints under consideration, or should leave them existentially quantified. The same question can be raised for the discharge of unproven goals in proof extraction. TCG offers extraction both with and without discharge, such that the language designer can choose the appropriate behaviour.

We can now re-formulate the `let` rule in this spirit: The context modifier for the second premise extracts the subproof of the first premise (which will not have any open goals in this section), and perform the forward reasoning. The forward application is written as a rule application, subproof extraction is indicated by angle brackets:

$$\forall\,(_{x,e,e',s,t})\ \frac{\begin{array}{l}\vdash e : s\\ -\,(:_1 = x);\ +\,[\texttt{let\_binding}[x]]\,(\langle 1\rangle) \vdash e' : t\end{array}}{\textbf{let}\ x\ =\ e\ \textbf{in}\ e' : t}\ (\texttt{let\_subproof})$$

Rule `let_subproof` contains a *rule reference* to the following auxiliary rule:

$$\forall\,(_{e,t})\ \frac{\vdash e : t}{\mathbf{y} : t}\ (\texttt{let\_binding}\ [\mathbf{y}]\ )$$

This rule has a *parameter* **y**, which can be instantiated with a term for each reference to rule `let_binding`, thus sharing terms with the referencing context. In the above application, the variable x, hence the `let`-bound identifier, will be shared with the rule, thus producing exactly the situation from (4.1.1) above.

We have seen that the context modifier + may specify the rule to be added in several forms. TCG generalizes these patterns to *rule expressions* (Section 3.1.3); evaluating a rule expression then yields a set of rules to be inserted into the context. Rule expressions are inductively defined by the following constructions:

- A rule reference, which can again be
  - An inline rule $[p]$ (by syntactic restriction without premises and quantifier).
  - *rule*[*parameters*] A reference to the named rule. The variables in the *parameters* have lexical binding, they are resolved against the bound variables of the containing rule.
- $\langle i$: *options*$\rangle$ Rule extraction of the subproof constructed for the ith premise of the rule. The referenced premise must be left of the one containing the rule expression. The *options* control quantification and discharge of open goals.
- *rule*(*rule expressions*) Forward resolution of the *rule*, given as a rule reference, against the arguments. Premises are resolved left to right, and extraneous premises remain in the result rule. A premise of the *rule* can be exempt from resolution by inserting an argument "–" in the corresponding argument position; it then remains as a premise of the result rule. Since each of the argument expressions yields a set of rules in general, all combinations are tried, and the result consists of the successful resolution steps.
- **use**(*term*) The *term* is output to a string, using formatting instructions (Sections 4.1.1 and 3.1.5). The result is taken as the name of a file, which must contain rules that have been saved by a previous run (Section 4.1.5 and 3.1.3).
- **environment** References all rules in the context of the rule application.

4.1.3.3 Polymorphic Let

Type inference for polymorphic let is defined by the following typing rule [Mil78, DM82]:

$$\frac{\Gamma \vdash e : s \quad \tilde{\alpha} = \mathbf{FV}(s) \setminus \mathbf{FV}(\Gamma) \quad \Gamma, x : \forall \tilde{\alpha}.s \vdash e' : t}{\Gamma \vdash \mathbf{let}\ x = e\ \mathbf{in}\ e' : t}$$

The bound variable x is assigned the type scheme $\forall\tilde{\alpha}.s$, which allows x to be used polymorphically, with a new instance of the type scheme at each reference. The instantiation takes place in the variable reference rule:

$$\frac{x : \forall\tilde{\alpha}.t \in \Gamma \quad t' = t[\tilde{s}/\tilde{\alpha}]}{\Gamma \vdash x : t'}\ (\text{var})$$

Consider again the problem of deriving the knowledge $x : \forall\tilde{\alpha}.s$ from the proof of $e : s$. In the previous Section 4.1.3.2, we have already established the creation of $x : s$ by rule extraction and forward application. It remains to find the quantified variables $\tilde{\alpha}$. Just as rule abstraction, this form of quantification is tightly integrated with TCG's notion of proof: The $\tilde{\alpha}$ are exactly the *inner variables* (Sections 1.2.3.5, 2.4.6, Remark 2.4.20) of the subproof at $e : t$. An inner variable is a free variable that occurs in the subproof and that does not occur in the remainder of the proof. During rule extraction, TCG can efficiently (Section 3.3.1.2) compute the inner variables and quantify the extracted rule over them. Forward application maintains the quantification information.

In the example of polymorphic `let`, the typing rule is then

$$\forall\left(_{x,e,e',s,t}\right)\ \frac{\begin{array}{l}\vdash e : s \\ -\ (:_1 = x);\ +\ [\texttt{let\_binding}[x]]\ (\langle 1 : [\forall]\rangle) \vdash e' : t\end{array}}{\mathbf{let}\ x\ =\ e\ \mathbf{in}\ e' : t}\ (\texttt{let\_poly})$$

Note that rule extraction specifies the option $\forall$, written with the `quantify` keyword in the textual input:

```
rule let_poly
forall(x,e,e',s,t)
  let(x,e,e') : t
if  e  : s
and e' : t
under -(:.1.= x) +let_binding[x](<1: [quantify]>)
```

The use of inner variables for quantification is motivated from three observations, which are discussed next. First, we pragmatically check that the inner variables coincide with those variables computed by the original (let) rule. Looking at a proof node where `let_poly` is applied, we see that no more than the desired variables will be quantified:

$$\frac{\begin{array}{c}\vdots\ (\text{proof of } e : s) \\ \Gamma \vdash e : s\end{array} \qquad \Gamma' \vdash e' : t}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e' : t}\ \texttt{let\_poly}$$

The variables in Γ are not inner variables of the left subproof, so they will never be included in the quantifier. For the converse, consider how the proof construction of $e : s$ proceeds: A (type) variable introduced in that subproof can only escape the subproof if it gets unified with some type in Γ – but then it would not be quantified in the standard rule as well, as it now appears in Γ.

A second motivation is found in the intended meaning of type schemes and polymorphic values. The scheme $e : \forall\tilde{\alpha}.s$ captures the fact that a proof for $e : s[\tilde{t}/\tilde{\alpha}]$ *could* be obtained by replacing the $\tilde{\alpha}$ throughout the deduction of $e : s$. The meta-theoretical soundness proofs formalize this intention in a standard sequence of Lemmata (see Section 2.4.1.2): First, one shows (by structural induction on the proof derivation) that $\Gamma\sigma \vdash e : s\sigma$ for any substitution σ and $\Gamma \vdash e : s$. If we restrict σ, such that $dom(\sigma) \cap \Gamma = \varnothing$, then also $\Gamma \vdash e : s\sigma$. The largest sensible $dom(\sigma)$ that satisfies the restriction is obviously $\mathbf{FV}(s) \setminus \mathbf{FV}(\Gamma)$. Thus the standard quantifier captures just these variables that may be replaced in s without changing Γ. Compare this reasoning to Tcg's inner variables: The existence of derivation $\Gamma\sigma \vdash e : s\sigma$ corresponds to a *copy of the subproof* at $e : s$, with substitution σ applied throughout. If we restrict $dom(\sigma)$ to the inner variables of the subproof, then σ does not affect the remainder of the proof – we can thus obtain arbitrary instances $e : s\sigma$, as long as $dom(\sigma)$ is restricted to the inner variable of the subproof.

A third motivation stems from standard proof-theoretical studies of natural deduction and sequent calculi. For natural deduction, the *eigenvariable condition* for $\forall$-introduction can be formulated as [Pra65, Section 2.A] (see also [Gen35, Section 2.21]):

$$\frac{A}{\forall x.A(x/a)}$$

where "a must not occur in any assumption on which A depends". By identifying the typing context Γ with the open assumptions of natural deduction [NP01, Section 1.3, Chapter 8], this formulation becomes akin to the standard (let) rule. For example, Negri and von Plato [NP01, Definition 8.1.2] formulate:

$$\frac{\begin{array}{c}\Gamma\\ \vdots\\ A(y/x)\end{array}}{\forall x.A}$$

"if y does not occur free in Γ or $\forall x.A$". Since the latter restriction can be avoided by choosing a suitable y, the restriction is the same as that on polymorphism.

On the other hand, sequent calculi suggest Tcg's view in formulating [Gen35, Section 1.22] (see also [TS00, Section 3.1]):

$$\frac{\Gamma \vdash \Theta, J(a/x)}{\Gamma \vdash \Theta, \forall x.J(x)}$$

Here, the "object variable [...] a [...] must not occur in the lower sequent of the inference figure". Allowing for a suitable renaming of free variables, this restriction captures Tcg's inner variables. Note, however, that such a "suitable renaming" would incur a performance penalty, while Tcg's more strict notion avoids creating the need for renaming in the first place, without sacrificing generality.

4.1.3.4 Recursion

The standard rule for a recursive `let` binding assumes that x has type s in the check that its definition e has type s:

$$\frac{\Gamma, x : s \vdash e : s \quad \tilde{\alpha} = \mathbf{FV}(s) \setminus \mathbf{FV}(\Gamma) \quad \Gamma, \forall\tilde{\alpha}.s \vdash e : t}{\Gamma \vdash \mathbf{letrec}\ x = e\ \mathbf{in}\ e' : t}$$

It can obviously be formalized in TCG by using an additional context modifier in the first premise. Mutual recursion can be obtained by defining tuples (Section 4.1.4) of values recursively (as in [CDDK86]), but this solution is too cumbersome for practical languages. TCG's formalization of mutual recursion is explained in Section 4.1.3.6, as it uses *exports* from the next Section 4.1.3.5 in an essential way.

4.1.3.5 Parallel Bindings

The basic typing features of MINIML [CDDK86] have now been covered. Practical languages usually extend the language for the ease of the programmer, very often allowing several instances of a construct where MINIML allows only a single occurrence. For example, a single `let` may bind several variables at the same time with the following syntax:

```
syntax
exp: "let" bind_group "in" exp --> let($2,$4)
| "let" error "in let expression"
bind_group: bind        --> $1 :: []
| bind "and" bind_group --> $1 :: $3
bind: ID "=" exp --> bind(id[$1],$3)
```

The abstract syntax for `bind_group` uses the special constructors "`::`" and "`[]`", which are read *cons* and *nil*. At the internal term level, they are not distinguished from other constructors, and they do not extend the notion of proofs (Chapter 2). However, TCG provides a few auxiliary functions that can facilitate the processing of *lists* of the form x_1`::`$(x_2$`::`$(\cdots$ `::[]`$))$. The binding groups of this section are a prototypical example application of the list constructs. (One other instance has already been seen in the formatter `@[|]` (Section 3.1.5).

The second new element of the syntax is `error`: When a syntax error is encountered, YACC starts removing items from its stack until a reduction with an `error` rule becomes possible. At this point, the shown production issues the given error message, printing the location information of the error automatically in the standard *filename*:*line number*:*message* format.

Although binding groups do not change the type theory of the language significantly, the type checker must nevertheless check the bindings one by one. In the TCG formulation, the constructed proof for the expression now takes the following form:

$$\frac{\begin{array}{c} e_1 : s_1 \quad \cdots \quad e_n : s_n \\ \vdots \\ \langle\text{check binding group}\rangle \end{array} \quad \Gamma, x_1 : s_1, \ldots x_n : s_n \vdash e : t}{\mathbf{let}\ x_1 = e_1 \ldots x_n = e_n\ \mathbf{in}\ e : t}$$

As a consequence, the reasoning about rule extraction in Section 4.1.3.2 and 4.1.3.3 does not apply immediately: The knowledge to be extracted does not appear at the first premise, but is scattered throughout its subproof. TCG provides *exports* as a straightforward extension of rule extraction to the above situation: Each node in the proof tree can be labeled with an identifier that can be referenced in rule extraction. The rule expression $\langle i : exp\rangle$ then extracts all subproofs labeled *exp* in the subproof of premise i. The search for exports in a subproof can be controlled: Only if a node *propagates* exports, its children are included in the search. Propagation is indicated with a second label [*]. The above proof tree thus changes to:

$$\frac{\begin{array}{c}[\mathit{bind}]e_1 : s_1 \quad \cdots \quad [\mathit{bind}]e_n : s_n \\ \vdots\ \text{(all nodes labeled [*])} \\ [*]\langle\text{check binding group}\rangle\end{array} \qquad \Gamma, \langle\text{extract exports } \mathit{bind}\rangle \vdash e : t}{\textbf{let } x_1 = e_2 \ldots x_n = e_n \textbf{ in } e : t}$$

Note that the considerations about inner variables do carry over directly: At the point of extraction, the inner variables of the *whole* subproof of the left premise can be selected for quantification, because which each copy of that subproof, also a fresh copy of the extracted subproof is obtained.

It remains to construct the elided subproof $\langle$check binding group$\rangle$. Essentially, the following two rules are needed to iterate over the list of bindings and create a goal for each binding:

$$\frac{\vdash b[\mathit{bind}] \qquad \vdash \texttt{bind\_group}(l)[*]}{\texttt{bind\_group}(b::l)} \qquad \frac{}{\texttt{bind\_group}(\texttt{[]})}$$

For practical languages, this approach is cumbersome and error prone: For every list in the abstract syntax tree, a similar pair of rules has to be provided to check the elements. TCG frees the user from that burden by generating the required rules automatically for *iteration premises*, which are indicated by an ellipsis "..." behind the usual premise. In the current example, the TCG rule is

$$\forall\,(b,bs,e,t,ts,bs')\ \frac{\begin{array}{c}\vdash bs : ts \textbf{ export } \texttt{bind}\ldots \\ +\,[\texttt{exp\_binding}]\,(\langle 1 : \texttt{bind}[\forall]\rangle) \vdash e : t\end{array}}{\textbf{let } bs \textbf{ in } e : t}\ (\texttt{let})$$

The first premise specifies that $bs : ts$ is to be proven for each element of bs separately; the corresponding types will be found in the list ts, which is, however, not referred to in the above rule. Each of generated premises is labeled with export *bind*, and all the intermediate proof nodes will be labeled [*] as propagating exports. The behaviour of iteration premises is defined by automatically generated, auxiliary rules – it does not require modifications to the definition of proofs (Chapter 2).

A single binding is checked canonically by the following rule. Note that the goal gathers all available information about the binding in a single place. This technique is important for the later extraction.

$$\forall\,(x,e,t)\ \frac{\vdash e : t}{x = e : t}\ (\texttt{bind})$$

The second premise then extracts and inserts all exports simultaneously. Like the previous rules for `let`, it uses forward reasoning with the following auxiliary rule to create the desired rules:

$$\forall\,(x,e,t)\ \frac{\vdash x = e : t}{x : t}\ (\texttt{exp\_binding})$$

To complete the introduction to iteration premises, here is the rule for recursive `let` from the TCG input. Note how by iteration premises the extension from single to multiple bindings requires only a minimal notational overhead; hence, the straightforward extension of the theory is paralleled by a straightforward extension of the implementation.

```
rule let
forall(b,bs,e,t,ts,bs')
  let(bs,e) : t
if bs : ts export bind ...
and e : t
under + exp_binding(<1: bind [quantify] >)
```

4.1.3.6 Recursive Bindings

Mutual recursion introduces a significant amount of overhead into type checkers: To check

$$x_1 = e_1 \mathinner{..} x_n = e_n$$

where each e_i may refer to each of the x_j, the ML type checker will first choose a new type variable α_j as a type for x_j, then add $x_1 : \alpha_1 \mathinner{..} x_n : \alpha_n$ to the context and proceed to check $e_1 : \alpha_1 \mathinner{..} e_n : \alpha_n$. Unification then instantiates the type variables as necessary. In languages with type annotations, the sequence $x_1 : s_1 = e_1 \mathinner{..} x_n : s_n = e_n$ must be traversed to gather the s_i beforehand. Generally speaking, the complication is that the simple recursive process of type checking is broken up, and a limited form of "look-ahead" has to be implemented.

In TCG, iteration premises again circumvent these problems. The `letrec` with mutual recursion is implemented by the following rules:

$$\forall\,(bs,e,t,x)\ \frac{\begin{array}{l}\vdash \textbf{bindgroup}(bs)\ \textbf{export}^{*}\\ -\,(:_1 = x \longrightarrow :_1 = x);\ +[\texttt{exp\_binding}]\,(\langle 1 : \texttt{bind}[\forall]\rangle)\\ \vdash e : t\end{array}}{\textbf{letrec}\ bs\ \textbf{in}\ e : t}\ (\texttt{letrec})$$

$$\forall\,(bs,ts,x)\ \frac{\begin{array}{l}\vdash bs : ts\ \textbf{export}\ \texttt{fwd!}\ldots\\ -\,(:_1 = x \longrightarrow :_1 = x);\ +[\texttt{exp\_binding}]\,(\langle 1 : \texttt{fwd}\rangle)\\ \vdash bs : ts\ \textbf{export}\ \texttt{bind}\ldots\end{array}}{\textbf{bindgroup}(bs)}\ (\texttt{bindgroup})$$

The main work is done in rule `bindgroup`, which checks the recursive bindings in the first premise of rule `letrec`, whose second premise requires that the types of the bindings are confined to one subtree in order to quantify correctly. We have already discussed the working of rule `bindgroup`: By iteration, each element in *bs* is checked and the goal is exported as *fwd*. Note that the types ts are actually fresh variables $\alpha_1 \mathrel{..} \alpha_n$, for they do not appear anywhere else in the rule. The *fwd* goals thus have the form $x = e : \alpha$; however, no proof obligation is associated with them — the proposition "x is associated with the newly chosen type variable α" is already true. Therefore, the premise is made a *solved* premise, which is indicated by the ! behind the export name. The purpose of solved premises is to state a fact, rather than create a new proof obligation. Like iteration premises, solved premises do not require modifications to proofs (Chapter 2): Only their context is modified to contain a single rule $\forall p.p$.

With these definitions, the following expression can be type checked; both `even` and `odd` are assigned type `int` $\rightarrow$ `bool` as expected:

```
letrec even = \x. if (eq_int x) 0 then true  else odd  ((sub x) 1)
   and odd  = \x. if (eq_int x) 0 then false else even ((sub x) 1)
in even 5
```

4.1.4 Tuples and Matching

An n-tuple is a heterogeneous sequence of n values. Tuples are constructed by writing a (non-singleton) sequence of expressions between a pair of parentheses.[3] Type checking a tuple expression means checking each of its components separately.

$$\forall\,(es,ts)\ \frac{\vdash es : ts \ldots}{(es) : (ts)}\ (\texttt{tuple})$$

Decomposition of tuple values is accomplished by a `match` expression of the form

$$\textbf{match}\ exp = pat\ \textbf{in}\ e'$$

The pattern *pat* is a nested tuple expression with variables at the leaves:

```
syntax
pat: ID                --> id[$1]
| "(" pats_opt ")"     --> tuple($2)
pats_opt: /* empty */ --> []
| pats                 --> $1
pats: pat              --> $1 :: []
| pat "," pats         --> $1 :: $3
```

To type check `match`, the pattern must have the same type as the expression. The pattern variables receive the type of the value found at their position within *exp*'s value. The following TCG rule uses exports to access the inferred types of the pattern variables and introduces a new predicate $:_p$ for type-checking patterns.

$$\forall\,(p,e,e',s,t)\ \frac{\begin{array}{l}\vdash e : s\\ \vdash p :_p s\ \textbf{export}^{*}\\ +\,\langle 2 : \texttt{patvar}\rangle \vdash e' : t\end{array}}{\textbf{match}\ e = p\ \textbf{in}\ e' : t}\ (\texttt{match})$$

[3]We ignore any ambiguities with parentheses in expressions for this introductory example.

Checking a linear pattern is a simple recursive process through the tuple constructors; the pattern variables are solved premises (see Section 4.1.3.6).

$$\forall\,(_{x,t})\ \frac{\vdash \texttt{id}[x] : t\ \textbf{export}\ \texttt{patvar!}}{\texttt{id}[x] :_p\ t}\ (\texttt{pat\_var})$$

$$\forall\,(_{ps,ts})\ \frac{\vdash ps :_p\ ts\ \textbf{export}^{*} \ldots}{(ps) :_p\ (ts)}\ (\texttt{pat\_tuple})$$

With these definitions, the following expression receives type $\texttt{int} \times \texttt{int} \times \alpha \to \alpha \times \texttt{int}$:

```
\x. match x = (u,v,w) in (w,((add u) v))
```

Records with Labeled Fields Record types $\{x_1 : t_1 .. x_n : t_n\}$ with structural matching can be implemented in the same manner, if the check for type-equality is made explicit. Even type *checking* with width-subtyping [Pie02, Chapter 15] is possible.

However, type-inference for records with structural subtyping (where a record can always be used where a record with fewer fields is expected) does not generalize straightforwardly [Wan91, Wri94, EST95a, AC96].

Alternatively, ML-style records can clearly be implemented within the given framework. Here, a label is assigned to exactly one record type by declaration [OCa03].

4.1.5 Saving and Loading Results

Many languages allow separate compilation of source files, hence their type checker has to save the typing information computed for one file to disk, and re-load it when processing the next file. A special case is the description of modules in separate interface and implementation files: Here, the interface file must be checked for consistency before the implementation is type checked and matched against the interface. Having identified rules as the primary pieces of typing knowledge before, we can devise a general mechanism to support the mentioned disk operations:

- When the type check of a file is complete, its proof is searched for exports, starting from its root node. The exported subtrees are then extracted as rules to be serialized to a file.
- The context modifiers (Section 3.1.3) are extended by a `load` facility, which loads some previously saved rules from a file. As the new feature represents merely a different form of rule constants, the formalism does not need to be modified.

We now extend the running example with simple `define` clauses that specify bindings to be recorded on disk. A `using` clause then allows to re-read previously saved rules.

```
syntax
top: "define" ID "=" exp     --> define(id[$2],$4)
top: "using" ID "{" tops "}" --> using(file[$2],$4)
```

The `define` clauses are checked by the following rule and marked as exported to prepare the saving. Because the conclusion of a rule cannot be exported, the auxiliary non-terminal $\texttt{define}'$ is introduced.

$$\forall\,(x,e,t)\ \frac{\vdash \textbf{define}'\ x : t\ =\ e\ \textbf{export}\ \texttt{define}}{\textbf{define}\ x\ =\ e : t}\ (\texttt{define})$$

$$\forall\,(x,e,t)\ \frac{\vdash e : t}{\textbf{define}'\ x : t\ =\ e}\ (\texttt{define'})$$

The `run` command in the action of non-terminal `file` (Section 4.1.1) initiates the actual extraction. It takes an optional parameter `save`, which is a list of triples; each element triple is processed in turn.

$$(\textit{export id}, \textit{forward rule}, \textit{file})$$

The *export id* designates the exports to be selected, and *file* names the destination file. (A predefined record `input` holds information on the name of the input file, such that the name of the output file can be composed.) Finally, *forward rule* allows a single forward resolution step[4] (Section 4.1.3.3) to discard any unused material before the rule is written.[5]

```
{! run ~save: [ ("define", "save_defined", (input.base^".rls")) ] $1 !}
```

The auxiliary rule `save_defined` discards the expression from a definition

$$\forall\,(x,e,t)\ \frac{\vdash \textbf{define}'\ x : t\ =\ e}{x : t}\ (\texttt{save\_defined})$$

The `using` clause reloads dumped rules from the context. The argument of `load` must be a format term (Section 4.1.1).

$$\forall\,(f,tops,tys)\ \frac{+\,\textbf{load}(f.rls) \vdash tops : tys \ldots}{\textbf{using}\ f\,tops : \textbf{void}}\ (\texttt{using})$$

Renaming all imported identifiers x to qualified identifiers $\texttt{f.}x$ can be easily effected by adding to the `load` rule expression a forward resolution step with the following rule:

$$\forall\,(x,s)\ \frac{\vdash x : s}{\textbf{f}\,.\,x : s}\ (\texttt{rename}\ [\textbf{f}]\ )$$

[4]One may allow a general rule expression, only the question of notation arises (Footnote 5).

[5]The actions must be programmed in the implementation language of TCG, which is currently OCaml [OCa03]. `~save:` indicates that the optional parameter `save` follows; the square brackets denote a list literal; and round brackets with the comma denote a tuple literal; `^` is string concatenation.

4.2 A Library of Language Constructs

TCG's fragment mechanism (Section 3.1) allows the formalization of a type system to be factored into independent pieces, such that parts shared between languages can be reused. This section describes the basic constituents of type systems of the subsequently formalized languages. As a general guideline, a construct has been included in the library if it is used more than once, or is expected to be of general interest. The description in this section intends to give an overview.

Each of the fragments identified for the library formalizes a particular language construct by the following four aspects:

- The input syntax is extended by new productions.
- The internal term representation is fixed.
- The relevant judgments are introduced.
- The (type-) checking requirements are given by rules.
- The external form for LaTeX output is defined.

The library is organized around language constructs, which have been divided into the following categories.

lex The lexical conventions of the example languages include identifiers and literals of various types.

exp Expressions form the basic elements of computation. Their distinguishing property is that they return a value, hence can be assigned a type. Expressions include function application and λ-abstraction, constants and variables. The functional `let` expressions are included here, rather than in the **decl** below, because they produce a value.

sm Statements are the basis of imperative languages with the common `if`, `while`, `for` constructs; blocks serve to delimit the scope of declarations contained in the statements. In contrast to expressions, statements do not produce a value.

top The top level of a file usually has a special meaning: Some declarations (or definitions) are allowed only here, and type checking is initiated for each top level component. Also, a test bed for incompletely formalized languages is provided by top level queries that check particular language constructs.

tyexp Type expressions in all but the most trivial languages are subject to well-formedness conditions, just as expressions are subject to well-typedness: At least, all the mentioned types and type constructors must have been defined before. Kinds, as "types of types" provide just this desired check for well-formedness.

decl Declarations tend to re-occur in language families, even though the particulars may be different. For example, imperative languages often allow variables to be declared within statement blocks, and the surrounding features, such as object-oriented constructs, do not matter.

conv Conversions change the representation of some value of a given to type, such that it has a different type. Probably the most frequently used conversion in imperative languages is that from an l-value to an r-value. As only the type checker can detect the necessity for conversions, they have been integrated into the library.

4.2.1 Design Guidelines

The fragments in the library each define a specific feature of type systems. Although they are largely independent, they interact in some points, notably the form of judgments: If some fragment B wishes to reuse a rule from fragment A, then the premise to be proven must obviously match the conclusion of the rule from A. A second link occurs in the use of forward rules (Section 4.1.3,3.1.3), whose premises must match the premises of fragment A, while their conclusions must match the exported premises of fragment B: Their role in this context is precisely to transform the internal premises of A to knowledge that can be used by B. To keep the library consistent, it is therefore necessary to establish a few general principles of designing these interfaces.

4.2.1.1 Compilation

In processing a program, there are several tasks that the type checker can perform conveniently, because the necessary information becomes available without further efforts: It can insert conversions whenever it needs to modify the type; it can resolve the binding of names and shadowing because each rule about a name is created at some specific binding; thus it can resolve (ad-hoc) overloaded identifiers just by keeping in each rule a unique name for each instance. The names are generated by a built-in predicate $\overset{\text{newopq}}{\mapsto}$. In any of these cases, the result of a type check is not only *success* or *failure*, but a new term that captures the specific choices made by the checker. Unlike the simplistic demonstration checkers in Section 4.1, the subsequent checkers will each be geared towards this form of compilation.

For the predicates involved, this means that they have clearly marked inputs and outputs. For example, the type check for an expression is not only

$$e : t \qquad \text{but} \qquad e \mapsto e' : t$$

This predicate is to be read as: "After modifying e to e', the latter can be assigned type t." (see also [HR95]). The choice of $\mapsto$ is arbitrary, but TCG provides the infix notation `=>` which makes the source code of rules more readable.

In general, the library consistently introduces predicates

$$x \mapsto_c x' :_c y$$

where c is a category of objects under consideration. For example, if x and x' are types and y is a kind, then $c = k$. The type checking judgement above drops the annotation t to ease the writing in this most frequent case.

4.2.2 Expressions

The expressions in this section are designed to emulate the basic features of expressions in existing languages. The languages considered range from functional (ML [MTHM97], OCaml [OCa03], Haskell [Je99]) to imperative and object-oriented ones (C/C^{++} [KR88, Str97], Java [GJS00]). Since the syntactic features differ greatly between these languages

Operators	Precedence	Associativity
`()` (function application)	100	left
`[]` (subscripting)		
`.` (member selection)		
`++`, `--`	90	(prefix/postfix)
`!`		
`-`, `+`		
`&`		
`*`		
`:>` (infix type cast)		
`&,|`	80	left
`*,/,%`	70	left
`+,-`	60	left
`=`, `==`, `!=`, `<>`, `<=`, `>=`, `<`, `>`	50	none
`&&`	40	left
`||`	30	left
`,`	20	left
`:=`	10	none
λ-abstraction	5	right

Figure 4.1: Precedence and Associativity of Operators

— for example, Haskell has an offside-rule that allows curly braces to be replaced by indentation — the expressions implemented here model only those features that have an impact on typing. To obtain readable example input, however, some infix mechanisms are provided nevertheless.

4.2.2.1 Syntax and AST

The syntax of expressions is generated by the non-terminal `exp`. The provided primitives include function application (with infix operators) and λ-abstraction. Because we wish to treat both functional and imperative languages, the development of expressions is required twice: Once for the curried function applications (`exp.curry`) and once for argument vectors (`exp.argvec`). The shared functionality resides in `exp` directly.

Expressions in both cases introduce precedences among operators as shown in Figure 4.1. The selection of the relative order follows that of C/C^{++} [Str91, Section 3.2].[6] Precedences in Tcg are given numerically, deviating from the Yacc convention of using the relative order of precedence declarations: With numeric specifications, the inclusion order of fragments is irrelevant.

[6]However, we have given the binary `&` and `|` (which in C++ are bitwise operators) a higher precedence than the comparison symbols, because this models more closely the division of expressions into *terms* and *formulae* in logic [Gal86].

The internal representation of application is

$$\texttt{app}(f, a)$$

where f is the applied operator and a is the argument, or list of arguments, respectively. Application of infix operators are likewise encoded by

$$\begin{gathered}\texttt{app}(\texttt{app}(\texttt{id}[op], a_1), a_2)\\ \texttt{app}(\texttt{id}[op], a_1\texttt{::}a_2\texttt{:: []})\end{gathered}$$

Here, the *op* is the operator name, as found in the input file. Prefix and postfix operators indicate their argument position with a "`.`", for example the prefix `*` is represented by `id["*."]`. A non-terminal `op_name` allows the user to write down operator names outside of expressions; for example `'*.` yields the opaque `id["*."]`.

Other identifiers are handled by the opaque `id[`*name*`]`. Literals for strings, integers and booleans are contained in `exp.str`, `exp.int`, and `exp.bool`, respectively. For example, the integer literals add a token and a new production:

```
tokens
INT=[0-9]+
end tokens
```

```
syntax
exp: INT --> int[$1]
```

4.2.2.2 Type checking

Type checking expressions consistently uses the predicate (see Section 4.2.1)

$$e \mapsto e' : t$$

The standard rule for function application in `exp.curry.apply` is simply:

$$\forall\,(f,f',a,a',s,t)\ \frac{\begin{array}{l}\vdash f \mapsto f' : s \to t\\ \vdash a \mapsto a' : s\end{array}}{f\ a \mapsto f'\ a' : t}\ (\texttt{apply\_std})$$

The version for argument vectors in `exp.argvec.apply` only adds an iteration premise to traverse the arguments.

$$\forall\begin{pmatrix}f,args,f',args',\\ formals,t\end{pmatrix}\ \frac{\begin{array}{l}\vdash args \mapsto args' : formals \ldots\\ \vdash f \mapsto f' : (formals) \to t\end{array}}{f(args) \mapsto f'(args') : t}\ (\texttt{apply\_std})$$

There is also a variation that creates independent sub-proofs for arguments and operator and combines the results at the end.

$$\forall\begin{pmatrix}f,args,args',f',\\ args',atys,rty\end{pmatrix}\ \frac{\begin{array}{l}\vdash [\text{branch}]args \mapsto args' : atys \ldots\\ \vdash [\text{branch}]f \mapsto f' : (atys) \to rty\end{array}}{f(args) \mapsto f'(args') : rty}\ (\texttt{apply\_branch})$$

The λ-abstraction with one argument for curried function calls is standard:

$$\forall\,(x,e,s,t)\;\frac{-\,(:_1 = x);\;+\,\left[x : s\right] \vdash e : t}{\lambda x.e : s \to t}\;(\texttt{lambda\_std})$$

To complement argument vectors, λ-abstraction in `exp.argvec.lambda` allows for multiple parameters. The iteration premise creates goals with fresh variables and solves them immediately (see Section 4.1.3.6).

$$\forall\,(xs,xs',e,e',s,t)\;\frac{\begin{array}{l}\vdash xs \overset{\text{newopq}}{\mapsto} xs' : s\;\textbf{export}\;\texttt{lambda\_bound\_name}!\ldots\\ +[\texttt{exp\_lambda\_bound\_name}]\,(\langle 1 : \texttt{lambda\_bound\_name}\rangle)\\ \vdash e \mapsto e' : t\end{array}}{\lambda xs.e \mapsto \lambda xs'.e' : (s) \to t}\;(\texttt{lambda})$$

The typing rules for literals use opaque patterns (see Section 4.1.2), for instance:

$$\forall\,(i)\;\texttt{int}[i] \mapsto i : \textbf{int}\;(\texttt{int})$$

The usual operator constants on integers and booleans are defined in `exp.intops` and `exp.boolops`. For example, the following rule declares the operator `>=` on the integers. Note that its translation has resolved a possible overloading of the operator.

$$\texttt{>=} \mapsto \texttt{ge\_int} : (\textbf{int}, \textbf{int}) \to \textbf{bool}\;(\texttt{geq\_int\_cnst})$$

4.2.3 Type Expressions and Kinds

As soon as user-defined type constructors are introduced, the consistency of parsed type expressions must be checked: All constructors must be defined and they must be applied to the right number of arguments. Kinds (e.g. [Jon95]) provide a simple framework for performing these checks: They parallel the assignment of types to expressions with an assignment of kinds to type expressions. Type constructors have kind $(\kappa_1 \mathinner{..} \kappa_n) \to \kappa_0$, where the κ_i are again kinds, while types without further arguments have kind $*$. For instance, the array constructor has kind $* \to *$, because it expects a single type and then yields a complete type. The fragment `kind.star` defines the constant `k_star` and its external formats. The kinding predicate employed in `tyexp` is

$$ty \mapsto_k ty' :_k \kappa$$

Each fragment in `tyexp` introduces a single type constructor, its syntax, kinding rules and external format. For example, the type constant `int` is introduced by

```
syntax
tyexp: "int" --> int
```

and its kind assignment is handled by

$$\textbf{int} \mapsto_k \textbf{int} :_k *\;(\texttt{int\_kind})$$

Likewise, an n-tuple is a type for each $n \in \mathbb{N}$. The file `tyexp.tuple` provides the syntax

```
syntax
tyexp: "<" tyexp_comma_list ">" --> tuple($2)
```

and checks the kind by iteration over the arguments

$$\forall\left(_{args, args'}\right) \frac{\vdash \mathit{args} \mapsto_k \mathit{args}' :_k * \ldots}{\langle \mathit{args} \rangle \mapsto_k \langle \mathit{args}' \rangle :_k *} \ (\texttt{tuple\_kind})$$

Further defined type constructors are arrays, boolean, references, strings and void.

4.2.4 Statements

Statements include the standard `if`, `while`, `for` constructs, blocks delimited by `begin`/`end`, and `return`. Furthermore, variables may be introduced within each block, they remain visible until the end of the block. The type checking rules are clear; they use a predicate

$$\mathit{sm} \mapsto_c \mathit{sm}'$$

where c stands for *compile*. A prototypical rule is that for `if-then`: It checks that the expression is a boolean value and then recursively compiles the `then` branch.

$$\forall\left(_{e, sm1, e', sm1'}\right) \frac{\begin{array}{c}\vdash e \mapsto e' : \mathbf{bool} \\ \vdash \mathit{sm1} \mapsto_c \mathit{sm1}'\end{array}}{\mathbf{if}\ e\ \mathbf{then}\ \mathit{sm1} \mapsto_c \mathbf{if}\ e'\ \mathbf{then}\ \mathit{sm1}'} \ (\texttt{ifthen})$$

A special case is the `return` statement: It retrieves the return type of the current environment using the special constant `return_type`. That constant can be easily inserted into the context after checking the header with the declared return type:

$$\forall\left(_{e, e', e', s, t}\right) \frac{\begin{array}{c}\vdash \text{return-type} : t \\ \vdash e \mapsto e' : t\end{array}}{\mathbf{return}\ e \mapsto_c \mathbf{return}\ e'} \ (\texttt{return})$$

Variable declarations are allowed wherever a statement is allowed:

```
syntax
sm: "var" id ":" tyexp --> vardecl($2,$4)
```

Variable declarations check the declared type and export an `sm_seq` stating that identifier `x` is an l-value of the stated type.[7] The special predicate $\overset{\text{newopq}}{\mapsto}$ expects to find an opaque value a at the left and a variable v at the right; it then instantiates v with a variant of a, which is certain not to be chosen at any other application of the predicate. Here, the

[7]We use the C^{++} notion that l-values are interchangeable with reference types.

predicate is used to generate unique names for the different variables, such that in the type checked result, all variable references are obvious.

$$\forall\,(_{x,x',ty,ty'})\ \frac{\begin{array}{l}\vdash x \overset{\text{newopq}}{\mapsto} x' \\ \vdash ty \mapsto_k ty' :_k * \\ \vdash x \mapsto x' : \&(ty')\ \textbf{export}\ \texttt{sm\_seq!}\end{array}}{\textbf{var}\ x : ty \mapsto_c \textbf{var}\ x' : ty'}\ (\texttt{vardecl})$$

The checker for sequences of statements then takes care to insert these exports for the remainder of the sequence: The context modifier noted at the iteration premise is executed for each iteration step; because only the local context is augmented, the declarations are automatically visible only until the end of the sequence.

$$\forall\,(_{sms,sms'})\ \frac{\vdash sms \mapsto_c sms'\ \textbf{export}^* \ldots + \langle 1 : \texttt{sm\_seq}\rangle}{sms \mapsto_c sms'}\ (\texttt{sm\_seq})$$

4.2.5 Top-level Constructs

The prototypical languages in this chapter have a very simple file structure: A file consists of a sequence of zero or more top-level elements. This simple structure is captured in `top.tops_wrap`

```
syntax
file: error "in file"
tops_wrap: tops --> tops($1)
tops: top  --> $1 :: []
| top tops --> $1 :: $2
```

This file also provides the trivial rule `tops0` for checking the empty file. However, it misses the initial `file:` production, because the `~save:` parameter (Section 4.1.5) varies for each language. File `top.tops` adds the trivial production

```
syntax
file: tops_wrap EOF {! run $1 !}
```

and the rule

$$\forall\,(_{ts})\ \frac{\vdash [\text{branch}]ts\ \textbf{export}^* \ldots}{\textbf{tops}(ts)}\ (\texttt{tops})$$

for checking very simple languages with completely independent top-level clauses.

4.2.6 Declarations

The part `decl` provides auxiliary features for declarations that appear in similar forms across a number of languages. Currently only the notion of formal parameters is implemented. File `decl.formals` thus introduces the syntax

```
syntax
formal__opt: --> []
| formal_ --> $1

formal_: formal --> $1 :: []
| formal ";" formal_ --> $1 :: $3

formal: id ":" tyexp --> formal($1,$3)
```

The consistency checking is done by a predicate

$$f \mapsto_f f'$$

The basic rules follow those for bindings (Section 4.1.3) and definitions (Section 4.1.5):

$$\forall\,(x,t,x',t')\ \frac{\vdash \texttt{formal'}(x \mapsto x', t \mapsto t')\ \textbf{export}\ \texttt{formal}}{x : t \mapsto_f x' : t'}\ (\texttt{formal})$$

$$\forall\,(x,x',t,t')\ \frac{\begin{array}{c}\vdash x \overset{\text{newopq}}{\mapsto} x' \\ \vdash t \mapsto_k t' :_k *\end{array}}{\texttt{formal'}(x \mapsto x', t \mapsto t')}\ (\texttt{formal'})$$

The predicate `formal'` serves to gather both the input and result of the $\mapsto_f$ predicate in one place, such that the exports can be translated properly. The rule

$$\forall\,(x,x',t,t')\ \frac{\vdash \texttt{formal'}(x \mapsto x', t \mapsto t')}{x \mapsto x' : t'}\ (\texttt{fwd\_formal\_to\_typing})$$

presents a checked formal parameter as a newly available value, while the following rule produces a newly available variable

$$\forall\,(x,x',t,t')\ \frac{\vdash \texttt{formal'}(x \mapsto x', t \mapsto t')}{x \mapsto x' : \&(t')}\ (\texttt{fwd\_formal\_to\_typing\_ref})$$

Records in imperative languages (without object-oriented extensions [JW85, KR88]; see Section 4.4 for the treatment of classes) are introduced as new types incomparable to any other type. In the following grammar productions, the auxiliary non-terminals for iterated field definitions are left out for conciseness:

```
syntax
recdef: "record" ID rdfields_opt "end" --> recdef(tid[$2],$3)
rdfield: ID ":" tyexp --> rdfield(recfield[$1],$3)
```

Checking a record definition involves choosing a unique name for the record type and checking the well-formedness of the fields' types. Note that iteration over the fields is only in the variables *fields* and *fields*$'$, the variables *name* and *name*$'$ remain fixed.

$$\forall\begin{pmatrix}\mathit{name},\mathit{fields},\\ \mathit{name}',\mathit{fields}'\end{pmatrix}\ \frac{\begin{array}{l}\vdash \mathit{name} \overset{\text{newopq}}{\mapsto} \mathit{name}' \\ \vdash \mathit{name} \mapsto_k \mathit{name}' :_k *\ \textbf{export}\ \texttt{reckind!} \\ \vdash \textbf{field}\mathit{name}(\mathit{name}') : \mathit{fields} \mapsto \mathit{fields}' \\ \quad \textbf{export}\ \texttt{recfield}\ldots[\mathit{fields},\mathit{fields}']\end{array}}{\textbf{record}\mathit{name}\mathit{fields}\textbf{end}}\ (\texttt{recdef})$$

Each field in the definition is checked by checking that the declared type has kind $*$ (Section 4.2.3).

$$\forall\begin{pmatrix}\scriptstyle name, name', f,\\ \scriptstyle ty, f', ty'\end{pmatrix}\ \frac{\begin{array}{c}\vdash f \overset{\text{newopq}}{\mapsto} f'\\ \vdash ty \mapsto_k ty' :_k *\end{array}}{\mathbf{field}\mathit{name}(\mathit{name}') : f : ty \mapsto f' : ty'}\ (\texttt{rdfield'})$$

The proofs for `recfield` are extracted using the following rule `fwd_recfield`. It asserts that the (unique) record type *rec*$'$ has a field f with unique name f', and that field is an l-value of type *ty*$'$.

$$\forall\begin{pmatrix}\scriptstyle rec, f, rec', f', ty,\\ \scriptstyle ty'\end{pmatrix}\ \frac{\vdash \mathbf{field}\mathit{rec}(\mathit{rec}') : f : ty \mapsto f' : ty'}{\mathit{rec}'.f \mapsto f' : \&(ty')}\ (\texttt{fwd\_recfield})$$

4.2.7 Conversions

Conversions allow the compiler to automatically change the representation of a value to satisfy some typing requirements. (Some authors prefer the term *coercions* for these automatic conversions.) Conversions are requested by the predicate

$$e : t \leq_c e' : t'$$

where e is an expression of type t and e' is an expression of type t'. The use of symbol $\leq$, rather than $\mapsto$, indicates the similarity with subtyping [Car93, Mit91, EST95a, JP99]. (See for example [BTCGS91, HR95] for the use of coercions as subtyping.)

The part `conv` of the library provides rules to solve these predicates. Fragment `conv.refl` establishes reflexivity of $\leq_c$ by

$$\forall\,(e,t)\ e : t \leq_c e : t\ (\texttt{conv\_refl})$$

Transitivity declared in `conv.trans`

$$\forall\,(e,e',e'',t,t',t'')\ \frac{\begin{array}{c}\vdash e : t \leq_c e' : t'\\ \vdash e' : t' \leq_c e'' : t''\end{array}}{e : t \leq_c e'' : t''}\ (\texttt{conv\_trans})$$

The fragment `conv.ref_deref` accesses l-values, which have reference type, by introducing the automatic dereference operator $*_a$ (`auto_deref`).

$$\forall\,(e,t)\ e : \&(t) \leq_c *_a(e) : t\ (\texttt{deref\_conv})$$

The rule `subsume` inserts the conversion predicate into a normal typing deduction:

$$\forall\,(e,e',e'',s,t)\ \frac{\begin{array}{c}\vdash e \mapsto e' : s\\ \vdash e' : s \leq_c e'' : t\end{array}}{e \mapsto e'' : t}\ (\texttt{subsume})$$

This rule resembles the following standard rule for subsumption in subtyping disciplines, which allows a value of type s to be assigned any supertype of s. Only the modification of e to e'' is peculiar to the conversion case.

$$\frac{\Gamma \vdash e : s \qquad s \leq t}{\Gamma \vdash e : t}$$

4.3 Imperative Languages

This section discusses prototypical elements of imperative languages. Because of the similarities, the name of Pascal has been chosen for the example language.

The type systems of many, if not all, imperative languages are distinguished from those of functional languages (Section 4.1) by their pragmatism: Because automatic coercions, the handling of l- and r-values, and overloaded operators on primitive types have become standard assets, and thus are expected by programmers, type inference is not a feasible approach. As type annotations are thus required for all variables and parameters, the type checking of expressions usually deals with monomorphic types without type variables. Where generic functions are allowed, such as in C^{++} [Str97], their instantiation must be determined by the types of the actual parameters alone, the return type is not considered. Where even this limited form of type inference would require major extensions to the type checker, explicit instantiation information is required. All of these features lead up to a single consequence: Type checking is a process local to single language constructs, as opposed for example to ML type inference, which takes into account all uses of a variable to determine its type.

Most typing features have already been treated in the library: The expressions in `exp.argvec` (Section 4.2.2) and statements in `sm` (Section 4.2.4) are geared toward imperative languages; the conversions in `conv` are sufficient for elementary purposes; Section 4.2.6 formalizes record types. It remains to assemble the parts into a self-contained language.

4.3.1 File Structure

The top-level file structure is a sequence of `top` elements (Section 4.2.5). The start nonterminal `file` saves the defined procedures for later reference (Section 4.1.5). The generic component `top.usefile` adds the corresponding inclusion mechanism for interface files.

```
syntax
file: tops_wrap EOF {!
    run ~save: [ ("procedure",
                  "fwd_procedure_to_typing",
                  input.base^".hdr") ] $1
!}
```

The auxiliary rule `fwd_procedure_to_typing` will be defined below.

The record definitions imported from `decl.record` (Section 4.2.6) are integrated as top-level components; when the check of a record definition is completed, the computed knowledge about its name and fields becomes available in the context:

$$\forall \binom{\mathit{name}, \mathit{fields},}{\mathit{tops}} \quad \frac{\begin{array}{l} \vdash \mathbf{record}\,\mathit{name}\,\mathit{fields}\,\mathbf{end}\ \mathbf{export}^{*} \\ +\,[\mathtt{fwd\_recfield}]\,(\langle 1 : \mathtt{recfield}\rangle)\,; \\ +\,\langle 1 : \mathtt{reckind}\rangle \vdash \mathbf{tops}(\mathit{tops})\ \mathbf{export}^{*} \end{array}}{\mathbf{tops}(\mathbf{record}\,\mathit{name}\,\mathit{fields}\,\mathbf{end}, \mathit{tops})}\ (\mathtt{tops\_recdef})$$

An example input file is shown in Figure 4.2 (Page 116).

```
procedure fac(i : int) : int
begin
  var n : int;
  n := 1;
  while i > 0 do
  begin
    n := n * i;
    i := i - 1
  end;
  return n
end

procedure main(j : int)
begin
  var k : int;
  k := j + 10;
  fac(k + 1)
end

record point
  x : int;
  y : int
end

procedure move(p : point; dx : int; dy : int)
begin
  p.x := p.x + dx;
  p.y := p.y + dy
end
```

Figure 4.2: Example Input for the PASCAL Checker

4.3.2 Procedures

Procedures use the following syntax, where the non-terminal `formal__opt` is defined in `decl.formals` (Section 4.2.6).

```
syntax
top: "procedure" id_or_op "(" formal__opt ")" ret_opt "begin" sm_seq "end"
  --> procedure($2,$4,$6,$8)
```

To keep the discussion concise, we will not consider mutually recursive procedures. The general mechanisms for handling mutual recursion has been discussed in Section 4.1.3.6, a second more elaborate example appears in classes with mutual references (Section 4.4). Nested procedures do not complicate the type checking significantly either: Shadowing of outer variables is the common case in typed λ-calculi (Section 4.1.1).

Type checking a procedure involves two steps: After checking the well-formedness of type expressions in the procedure head, the procedure body is checked under new assumptions for the formal parameters and the return type. Here, type expressions are checked by computing kinds (Section 4.2.3). Following the technique from Section 4.2.6, the auxiliary predicate predicate `procedure'` contains both input and result of the type check in a single place, such that all relevant information is available for determining the procedure's type by rule extraction.

$$\forall\begin{pmatrix} \mathit{name},\mathit{name}', \\ \mathit{formals}, \\ \mathit{fnames}',\mathit{ftys}', \\ \mathit{ret},\mathit{ret}',\mathit{body}, \\ \mathit{body}' \end{pmatrix}\ \frac{\begin{array}{c}\vdash \textbf{procedure}'\ \mathit{name}/\mathit{name}'(\mathit{fnames}' : \mathit{ftys}') : \mathit{ret}' \\ \textbf{begin}\ \mathit{body}/\mathit{body}'\ \textbf{end}\ \textbf{export}\ \texttt{procedure}\end{array}}{\textbf{procedure}\ \mathit{name}(\mathit{formals}) : \mathit{ret}\ \textbf{begin}\ \mathit{body}\ \textbf{end}}\ (\texttt{procedure})$$

$$\forall\begin{pmatrix} \mathit{name},\mathit{name}', \\ \mathit{formals}, \\ \mathit{fnames}',\mathit{ftys}', \\ \mathit{ret},\mathit{ret}',\mathit{body}, \\ \mathit{body}' \end{pmatrix}\ \frac{\begin{array}{l}\vdash \mathit{name} \overset{\text{newopq}}{\mapsto} \mathit{name}' \\ \vdash \mathit{formals} \mapsto_f \mathit{fnames}' : \mathit{ftys}'\ \textbf{export}^* \ldots \\ \vdash \mathit{ret} \mapsto_k \mathit{ret}' :_k * \\ +[\texttt{fwd\_formal\_to\_typing\_ref}]\,(\langle 2 : \texttt{formal}\rangle)\,; \\ \quad + \big[\text{return-type} : \mathit{ret}'\big] \vdash \mathit{body} \mapsto_c \mathit{body}'\end{array}}{\begin{array}{c}\textbf{procedure}'\ \mathit{name}/\mathit{name}'(\mathit{fnames}' : \\ \mathit{ftys}') : \mathit{ret}'\ \textbf{begin}\ \mathit{body}/\mathit{body}'\ \textbf{end}\end{array}}\ (\texttt{procedure'})$$

Rule `procedure'` chooses a new, unique name for the procedure, and infers kinds for the type formal parameters and return type. Finally, the procedure body is compiled after adding the formal parameters and return type to the context (Section 4.2.4). (The rule `fwd_formal_to_typing` interfaces (in the sense of Section 4.2.1) with `decl.formal` and is thus defined in that fragment (Section 4.2.6).) Note that *fnames*′ is a sequence of unique names for the parameters (Section 4.2.6), such that all references in *body*′ be resolved.

In remains to integrate procedures to the context in which they appear. The rule `top_procedure` checks a procedure element of the file and adds its type information to the context of the remaining top-level elements.

$$\forall \binom{name, formals,}{ret, body, tops} \frac{\begin{array}{l}\vdash [\text{branch}]\textbf{procedure}\ name(formals) \\ \ :ret\ \textbf{begin}\ body\ \textbf{end}\ \textbf{export}^* \\ \textbf{expand} \\ +[\texttt{fwd\_proc\_to\_typing}]\ (\langle 1 : \texttt{procedure}\rangle) \\ \ \vdash \textbf{tops}(tops)\ \textbf{export}^*\end{array}}{\begin{array}{l}\textbf{tops}(\textbf{procedure}\ name(formals) \\ :ret\ \textbf{begin}\ body\ \textbf{end}, tops)\end{array}} \ (\texttt{top\_procedure})$$

It uses the auxiliary rule `fwd_procedure_to_typing` to extract the required information from the `procedure'` predicate. The same rule is also used for saving the results of type inference in the `file:` action (Section 4.3.1).

$$\forall \begin{pmatrix} name, name', \\ formals, \\ fnames', ftys', \\ ret, ret', body, \\ body' \end{pmatrix} \frac{\begin{array}{l}\vdash \textbf{procedure}'\ name/name'(fnames' : \\ \ ftys') : ret'\ \textbf{begin}\ body/body'\ \textbf{end}\end{array}}{name \mapsto name' : (ftys') \rightarrow ret'} \ (\texttt{fwd\_proc\_to\_typing})$$

4.3.3 Conversions

Conversions are found in imperative languages in two areas: First, dereferencing of variables is a very frequent operation, such that a conversion of an l-value (or reference type) to an r-value is inserted automatically where necessary. Second, the primitive atomic data types are not distinguished strictly, an extreme example being C [KR88]. We will be concerned with the dereferencing in this section and treat an example of subtyping in Section 4.4.

The typing rules in the library (Section 4.2) have been designed to reflect standard situations from the literature, and these do not provide for conversions. However, Section 4.2.7 defines the rule `subsume`:

$$\forall (e, e', e'', s, t) \frac{\begin{array}{l}\vdash e \mapsto e' : s \\ \vdash e' : s \leq_c e'' : t\end{array}}{e \mapsto e'' : t} \ (\texttt{subsume})$$

An unrestricted application of this rule leads to cycles in the deduction and non-termination – an explicit restriction of the proof structure (Section 3.1.4) becomes necessary. The restriction is specified by a tree grammar [CDG+99] for the proof tree. The production

$$r : c \longrightarrow p_1 \ldots p_n$$

attaches the non-terminal c to the conclusion of rule r and non-terminal p_i to its ith premise. The symbol $*$ is a special default non-terminal that is attached to all conclusions and premises not mentioned explicitly in the grammar .

The non-terminal *sub* labels those instances of the predicate $\cdot \mapsto \cdot : \cdot$ where a conversion may be applied. The following productions locate these instances in the typing rules from

Section 4.2:

$$\begin{array}{rcccl} \texttt{apply\_branch} & : & * & \longrightarrow & \mathit{sub}, * \\ \texttt{recfield\_acc} & : & * & \longrightarrow & \mathit{sub}, * \\ \texttt{return} & : & * & \longrightarrow & *, \mathit{sub} \end{array}$$

Then the rule `subsume` must be tagged for resolving these instances

$$\texttt{subsume} \;:\; \mathit{sub} \;\longrightarrow\; *, *$$

The above instructions create goals $e : t \leq_c e' : \alpha$ with a variable α whenever the target type of the conversion has not been instantiated by other rule applications. Solving these goals enumerates all the possible target types, even though only one will be picked in the end. An ad-hoc solution is to order the premises in the typing rules such that the target type is ground whenever a conversion goal is created. TCG provides a more general solution by the *defer* mechanism (Section 3.1.4): We can specify that a conversion where the target type is still a variable will not be resolved. The selector (Section 3.1.1) used for describing the deferred goals has been explained in Section 4.1.1. Here, the second argument of `<=c` must be an application of `:`, whose second argument is a variable.

$$\textbf{defer}\ \mathit{conv} : \texttt{<=c}_2\texttt{:}_2\langle \mathit{var}\rangle$$

4.3.4 Overloading

Imperative languages usually provide overloaded versions for operators, at least for primitive numeric types: Using the typing information on `a` and `b`, the compiler determines whether `a+b` is to invoke integer or floating point addition, and the generated code reflects the chosen operator instance directly. The general task of the compiler is therefore to replace each overloaded function symbol in the syntax tree with a non-overloaded function symbol such that the type constraints between the actual and formal parameters are satisfied.

The possible choices for unique functions interact with available conversions: If there is a conversion from `int` to `real`, then `a+b` may denote either integer or real addition for `a`, `b : int`. Therefore, overload resolution is not unique without imposing further conditions, and several choices are possible:

- C^{++} [Str97] insists that overloading is resolved locally at each operator application. The possible operator instances are selected by the number and types of the actual arguments, taking into account the defined conversions. Among these operators, a single *best match* is sought with respect to the number and type of conversions. (The details of the process are intricate [ISO98] and are governed by traditions and the expected behaviour in specific cases.) If no best match is found, compilation fails. A similar strategy is chosen by Java [GJS00].
- Ada [Bar95, GR80, Bak82] resolves overloading by taking into account the context of the operator application as well: An expression is checked recursively, delivering *all* possible choices of unique function symbols. At a surrounding operator application, those resolutions not matching any formal parameter type are discarded. After proceeding in this manner for an entire expression, a single possible resolution must remain.

TCG inherently supports overload resolution by non-deterministic proof search: The standard application rule

$$\forall \binom{f, args, f', args',}{formals, t} \; \frac{\vdash args \mapsto args' : formals \ldots \quad \vdash f \mapsto f' : (formals) \to t}{f(args) \mapsto f'(args') : t} \; (\texttt{apply\_std})$$

enumerates all possible resolutions for the arguments in turn and finally tries to apply all declared operators f, assuming that f' is a unique name. Unfortunately, argument i will thus be re-checked for each combination of resolutions for arguments 1 to $i-1$, even if it does have a single resolution only: Deviating from the Baker-Algorithm for Ada [Bak82], the arguments are not checked independently.

The *branching* mechanism (Section 3.3.4.5) solves this efficiency problem by starting independent *sub-searches* for several goals. The rule `apply_branch` from `exp.argvec.apply`, which PASCAL uses, branches for the individual arguments and operators. An *expand* directive tells TCG to generate at this point all possible combinations of result proofs from the preceding branches; if no *expand* is given, it is appended to the list of premises. Thus in the following rule, the arguments are checked independently and the implicit *expand* determines all possible combinations with operators. Incidentally, any deferred conversions (Section 4.3.3) between actual and formal argument types will become ground at this point and will be resolved.

$$\forall \binom{f, args, args', f',}{args', atys, rty} \; \frac{\vdash [\text{branch}]\, args \mapsto args' : atys \ldots \quad \vdash [\text{branch}] f \mapsto f' : (atys) \to rty}{f(args) \mapsto f'(args') : rty} \; (\texttt{apply\_branch})$$

The option *unique* forces TCG to check that a single result is obtained for the branch.

$$\forall \,(e, e', t) \; \frac{\vdash [\text{branch} : unique]\, e \mapsto e' : t}{e \mapsto_c e'} \; (\texttt{sm\_exp})$$

TCG thus can be made to simulate the Baker-Algorithm [Bak82] directly, and with minimal notational overhead over languages without overloading. However, TCG does not currently allow a set of preferences between solutions to be given, thus a C^{++}-like regime for local resolution cannot be implemented: Preferences are necessarily outside of the usual proof search for goals, because they need to consider *several* proofs at once. A reasonable point of extension would be the *expand* directive: If it were equipped with some predicate for comparing the result proofs after expansion, it could choose between alternatives. However, the details of this mechanism can only be fixed after studying several example applications, which is deferred to future work.

4.4 Object-Oriented Languages

Object-oriented languages support a classical set of features [Boo91, Section 2.2][AC96, Part I][Str97, Chapters 5 and 6]. The language CLASS described in this section captures the following selection.

Objects An *object* consists of *(member) variables* [Str97] which together form the *state* of the object. Besides the variables, an object has a set of *methods* that operate on its state. State and methods are inseparable from the object and they can be accessed only by designating a single object. (We leave aside static methods [GJS00][Str97], their treatment is similar to that of procedures in Section 4.3.) Member variables are mutable variables in the sense of Section 4.3. In the subsequent material, we will call the member variables and methods collectively the *fields* of an object.

Methods Within the methods of one object, the object itself is accessible as a compound data structure *this* (or *self*). *this* can be understood as an implicit parameter passed to methods whenever they are invoked. For brevity, a field x in *this* can also be accessed by x alone; the type checker inserts the implicit indirection $this.x$.

Encapsulation An object declares fields as either *public* or *private*: Public fields constitute the object's interface, they are accessible from the outside; private fields are accessible only to the object's own methods.

Classes Class-based languages [AC96] add to the notion of objects that of *classes*. A class is a template for objects with the same set of fields, and the class acts as a type for the object (but see e.g. [CHC89, LW94, SGM02]). A new object of a class is created (or initialized [Str97]) by calling one of the class's *constructors*, which are methods whose *this* references uninitialized memory.

Inheritance A class may *inherit* from a *super class*, in which case the class receives all the fields and methods of the super class. The class is then called a *derived class* (of the super class). It may add new fields and methods. It may also *override* methods defined in the super class, meaning that the implementation of the method is replaced by a new one. In a method invocation, the method found in a particular instance object will be called, regardless of static typing. Concerning encapsulation, a derived class may access also the private fields of its super classes. (A more detailed specification by *protected* fields [Str97] could be accomplished within the framework presented in this section.)

Subtyping Inheritance induces a *subtype relation*: A value of a derived class is always acceptable where a value of a super class is expected. This relation is transitive. For practical reasons, object-oriented languages also allow *down-casts* that force conversion from a super-class to a sub-class.

Mutual Recursion This feature is implicit in standard presentations of object models: They assume that the definitions of classes may refer freely to (the interfaces of) other defined classes. It is listed here because it will have some impact on the overall structure of the type-check to be executed.

4.4.1 Remark. A wholly different tradition of object-oriented programming has been developed for functional languages (see for example [Red88, Wan91, Rém91, CM94, EST95b]): An object is perceived as a record value, where the *this* reference is implemented by fixpoint operators [Red88], and methods become λ-abstractions. In this setting, the typing properties of objects can be derived from the elementary rules for functions, records and fixpoints. Classes are not needed in these languages: Objects have (recursive) record types; subtyping between objects is likewise subtyping of records [Pie02]. Type-inference can be accomplished in two ways: First, row variables [Wan91, Rém91, OCa03] enable ML-style inference by an extended unification procedure. Second, the type inference is expressed as a constraint satisfaction problem to be solved [EST95b, EST95a].

4.4.1 Syntax

The syntax for classes follows C^{++} and JAVA, but adapts the choice of keywords to those of the library (Section 4.2). Figures 4.3 and 4.4 show example code. The top-level structure of a file consists of a set of classes, which may refer to each other (see *mutual recursion* above). Each class declares a name, a super class (or OBJECT by default) and its fields:

```
syntax
top: "class" ID inherit_opt "begin" field_ "end" --> cls(tid[$2],$3,$5)
```

Fields are either variables or methods:

```
syntax
field_cnt: "var" id ":" tyexp --> var($2,$4)
```

```
syntax
field_cnt: override_opt "method" id "(" formal__opt ")" ret_opt
           "begin" sm_seq "end" --> method($1,$3,$5,$7,$9)
```

Following Cardelli [AC96, Section 12.2], overriding of methods must be stated explicitly:

```
syntax
override_opt: --> introduce
 | "override" --> override
```

The convention of C^{++}/JAVA, where overriding methods are identified by matching the signature against inherited methods could be implemented by a search as well; it does not contribute to the typing mechanisms, however.

4.4.2 Type Checking

Mutual recursion at the file level forces the type checker to proceed in three phases:

1. Gather the class names to enable subsequent kind checks.
2. Check variables and method headers for well-formed types.
3. Type check the method bodies.

Note that mutual recursion across files can be established by saving the results of Passes 1 and 2 separately and reloading that information for each involved file.

```
class cell
begin
   private var cnt : object
   public method get() : object
   begin
      return cnt
   end
   public method set(x : object)
   begin
     cnt := x
   end
  public constructor()
  begin
    cnt := null
  end
  public constructor(c : object)
  begin
    cnt := c
  end
end

class backup inherits cell
begin
  private var backup : object
  public method restore()
  begin
    cnt := backup
  end
  public override method set(x : object)
  begin
    backup := x;
    super.set(x)
  end
  public constructor(c : object)
  begin
    super(c);
    backup := null
  end
end
```

Figure 4.3: Input for Checker Class (I)

```
class integer
begin
   private var i : int
   public method get() : int
   begin
     return i
   end
   public method set(j : int)
   begin
     i := j
   end
   public constructor(j : int)
   begin
     i := j
   end
end

class main
begin
  public method main()
  begin
    var cell : cell;
    cell := new backup(new integer(4));
    var i : integer;
    i := cell.get() :> integer;
    cell.set(new integer(i.get() + 1))
  end
end
```

Figure 4.4: Input for Checker Class (II)

The rule `tops` to be used for a parsed file displays the three phases clearly in three separate goals:

$$\forall\,(\mathit{tops})\ \frac{\begin{array}{l}\vdash [\text{branch}]\mathbf{top}_I(\mathit{tops})\ \mathbf{export}^*\ldots\\ \mathbf{expand}\\ +\langle 1:\texttt{cls\_name}\rangle \vdash [\text{branch}]\mathbf{top}_{II}(\mathit{tops})\\ \ \mathbf{export}^*\ldots +\langle 1:\texttt{cls\_inherits}\rangle;\\ \ +[\texttt{fwd\_method\_in\_cls}]\,(\langle 1:\texttt{cls\_method}\rangle)\\ \mathbf{expand}\\ \ +\langle 1:\texttt{cls\_name}\rangle;\\ \ \ +[\texttt{fwd\_method\_in\_cls}]\,(\langle 3:\texttt{cls\_method}\rangle)\,;\\ \ \ +[\texttt{fwd\_var\_in\_cls}]\,(\langle 3:\texttt{cls\_var}\rangle)\,;\\ \ \ +[\texttt{fwd\_ctor\_in\_cls}]\,(\langle 3:\texttt{cls\_ctor}\rangle)\,;\\ \ \ +\langle 3:\texttt{cls\_name}\rangle;\ +\langle 3:\texttt{cls\_inherits}\rangle\\ \ \ \vdash [\text{branch}]\mathbf{top}_{III}(\mathit{tops})\ \mathbf{export}^*\ldots\end{array}}{\mathbf{tops}(\mathit{tops})}\ (\texttt{tops})$$

Each top-level goal augments the context with the newly proven facts from the preceding phases. Then the goal is proven separately for each class in the file by an iteration premise.

4.4.3 Predicates

Following the lead of Section 4.2.1, we state the predicates used in the subsequent development beforehand. Fields in classes are described by their type and access rights; by slight generalization, the predicate

$$\mathbf{in}\ c(a) : p$$

states that class c contains a declaration, as stated by predicate p, which is accessible as $a \in \{\texttt{public}, \texttt{private}\}$. The second pass will add such a predicate to the global context for each field found during the check.

Encapsulation of fields is handled by the following predicate:

$$\mathbf{access}\ c\ \mathbf{as}\ a$$

This predicate states that the fields with access $a \in \{\texttt{public}, \texttt{private}\}$ within class c are visible. The context of a goal will contain all the ACCESS predicates to describe its access rights to fields.

Inheritance of derived class c from super class s is recorded introducing the following predicate into the context for Pass 3:

$$c\ \mathbf{inherits}\ s$$

4.4.4 Checking the Interface

Pass 1 consists in the following single rule which only exports a fact about the class name (see Section 4.1.3.6):

$$\forall\begin{pmatrix}\mathit{cls},\mathit{inherits},\\ \mathit{fields},\mathit{inherits}'\end{pmatrix}\ \frac{\vdash \mathit{cls} \mapsto_k \mathit{cls} :_k * \mathbf{export}\ \texttt{cls\_name!}}{\mathbf{top}_I(\mathbf{class}\ \mathit{cls}\ \mathbf{inherits}\ \mathit{inherits}\ \mathit{fields}\ \mathbf{end})}\ (\texttt{top\_I\_cls})$$

Pass 2 checks that all types in inheritance declarations, variables and methods are known by assigning the kind $*$ to each of them: (Section 4.2.3).

$$\forall\left(\begin{array}{l} \mathit{cls},\mathit{inherits}, \\ \mathit{fields},\mathit{inherits}' \end{array}\right)\ \frac{\begin{array}{l} \vdash \mathit{inherits} \longmapsto_k \mathit{inherits}' :_k * \\ \vdash \mathit{cls}\ \mathbf{inherits}\ \mathit{inherits}'\ \mathbf{export}\ \texttt{cls\_inherits}! \\ \vdash \mathbf{field}_{II}^{\mathit{cls},\mathit{inherits}'}(\mathit{fields})\ \mathbf{export}^{*} \ldots [\mathit{fields}] \end{array}}{\mathbf{top}_{II}(\mathbf{class}\ \mathit{cls}\ \mathbf{inherits}\ \mathit{inherits}\ \mathit{fields}\ \mathbf{end})}\ (\texttt{top\_II\_cls})$$

Methods and variables are checked like procedures (Section 4.3), but the bodies are left out. The following rule displays the similarity, using the `formals` from Section 4.2.6 again.

$$\forall\left(\begin{array}{l} \mathit{cls},\mathit{acc},\mathit{name}, \\ \mathit{name}',\mathit{formals}, \\ \mathit{fnames}',\mathit{ftys}', \\ \mathit{ret},\mathit{ret}' \end{array}\right)\ \frac{\begin{array}{l} \vdash \mathit{formals} \longmapsto_f \mathit{fnames}' : \mathit{ftys}' \ldots \\ \vdash \mathit{ret} \longmapsto_k \mathit{ret}' :_k * \end{array}}{\begin{array}{l} \mathbf{method}_{II}^{\mathit{cls},\mathit{acc}}\ \mathit{name}/\mathit{name}', \\ \mathit{formals} \mapsto \mathit{fnames}',\mathit{ftys}' : \mathit{ret},\mathit{ret}' \end{array}}\ (\texttt{method\_II'})$$

Introduced and overridden methods create different proof obligations: For new fields, a fresh name must be chosen, such that the field can be uniquely identified within the class. Note that for example a C^{++} compiler would at this point assign a new index in the virtual table.

$$\forall\left(\begin{array}{l} \mathit{cls},\mathit{inh},\mathit{acc}, \\ \mathit{name},\mathit{formals}, \\ \mathit{ret},\mathit{body}, \\ \mathit{name}', \\ \mathit{fnames}',\mathit{ftys}', \\ \mathit{ret}' \end{array}\right)\ \frac{\begin{array}{l} \vdash \mathit{name} \overset{\text{newopq}}{\mapsto} \mathit{name}' \\ \vdash \mathbf{method}_{II}^{\mathit{cls},\mathit{acc}}\ \mathit{name}/\mathit{name}'_{\mathit{cls}}\,\mathit{cls}, \\ \ \mathit{formals} \mapsto \mathit{fnames}',\mathit{ftys}' : \mathit{ret},\mathit{ret}' \\ \ \mathbf{export}\ \texttt{cls\_method} \end{array}}{\begin{array}{l} \mathbf{field}_{II}^{\mathit{cls},\mathit{inh}}(\mathbf{field}_{\mathit{acc}}\mathbf{introduce\ method}\ \mathit{name} \\ (\mathit{formals}) : \mathit{ret}\ \mathbf{begin}\ \mathit{body}\ \mathbf{end}) \end{array}}\ (\texttt{field\_II\_meth\_intr})$$

For overridden methods the old declaration in the super class must be found to establish the unique name of the method. Note that the goal `method_II'` is not exported, as the method is already known in the base class.

$$\forall\left(\begin{array}{l} \mathit{cls},\mathit{inh},\mathit{acc}, \\ \mathit{name},\mathit{formals}, \\ \mathit{ret},\mathit{body}, \\ \mathit{name}', \\ \mathit{fnames}',\mathit{ftys}', \\ \mathit{ret}',\mathit{acc}' \end{array}\right)\ \frac{\begin{array}{l} \vdash \mathbf{method}_{II}^{\mathit{cls},\mathit{acc}}\ \mathit{name}/\mathit{name}', \\ \ \mathit{formals} \mapsto \mathit{fnames}',\mathit{ftys}' : \mathit{ret},\mathit{ret}' \\ \vdash \mathbf{in}\ \mathit{inh}(\mathit{acc}) :\ \mathit{name} \mapsto \mathit{name}' : (\mathit{ftys}') \to \mathit{ret}' \end{array}}{\begin{array}{l} \mathbf{field}_{II}^{\mathit{cls},\mathit{inh}}(\mathbf{field}_{\mathit{acc}}\mathbf{override\ method}\ \mathit{name} \\ (\mathit{formals}) : \mathit{ret}\ \mathbf{begin}\ \mathit{body}\ \mathbf{end}) \end{array}}\ (\texttt{field\_II\_meth\_ovr})$$

Constructors receive the same treatment as methods, and a special name `ctor` to distinguish them from ordinary fields.

$$\forall\left(\begin{array}{l} \mathit{cls},\mathit{acc},\mathit{name}', \\ \mathit{formals}, \\ \mathit{fnames}',\mathit{ftys}' \end{array}\right)\ \frac{\begin{array}{l} \vdash \texttt{ctor} \overset{\text{newopq}}{\mapsto} \mathit{name}' \\ \vdash \mathit{formals} \longmapsto_f \mathit{fnames}' : \mathit{ftys}' \ldots \end{array}}{\mathbf{constructor}_{\mathit{name}'}^{\mathit{cls},\mathit{acc}}\mathit{formals} \mapsto \mathit{fnames}' : \mathit{ftys}'}\ (\texttt{ctor\_II'})$$

4.4.5 Checking the Method Bodies

Pass 3 of the type check does the main work: For each class and each method, the method's body is type checked. In this pass, the encapsulation attributes **private/public**, the inheritance relation and the access to `this` become relevant. As these ingredients remain unchanged within each class, the rule for classes can setup the context accordingly:

$$\forall \begin{pmatrix} cls, acc, \\ inherits, \\ inherits', fields, \\ inherits, \\ inherits' \end{pmatrix} \frac{\begin{array}{l} \vdash \mathit{inherits} \mapsto_k \mathit{inherits}' :_k * \\ + \texttt{decl\_this}[\mathit{cls}];\ + \texttt{open\_this}; \\ + \texttt{decl\_super}[\mathit{inherits}'];\ + \texttt{allow\_private}[\mathit{cls}] \\ \vdash \mathbf{field}_{III}^{cls, inherits'}(\mathit{fields}) \ldots [\mathit{fields}] \end{array}}{\mathbf{top}_{III}(\mathbf{class}\ \mathit{cls}\ \mathbf{inherits}\ \mathit{inherits}\ \mathit{fields}\ \mathbf{end})} \ (\texttt{top\_III\_cls})$$

The context modifier deserves attention: The auxiliary rules

$$\mathbf{this} \mapsto \mathbf{this} : \mathbf{cls}\ (\texttt{decl\_this}\ [\mathbf{cls}]\ ) \qquad \mathbf{super} \mapsto \mathbf{super} : \mathbf{cls}\ (\texttt{decl\_super}\ [\mathbf{cls}]\ )$$

define the `this` keyword as referring to the current object, and the `super` keyword as referring to the same object, but with the static type of the immediate super class. The rule

$$\forall \begin{pmatrix} cls, acc, this', x, \\ x', t \end{pmatrix} \frac{\begin{array}{l} \vdash \mathbf{this} \mapsto \mathit{this}' : \mathit{cls} \\ \vdash \mathbf{in}\ \mathit{cls}(\mathit{acc}) :\ x \mapsto x' : t \end{array}}{\texttt{id}[x] \mapsto \mathit{this}'.x' : t} \ (\texttt{open\_this})$$

allows the abbreviation of field *this.f* as just *f*; note that the implicit reference is inserted in the compiled (Section 4.2.1.1) version of the expression.

Finally, the access rights must be adjusted: The private parts of the current class (and all of its super classes, see Section 4.4.7) are open to all of the class's methods. Therefore, the corresponding **access** predicate is inserted to the context:

$$\mathbf{access\ cls\ as\ private}\ (\texttt{allow\_private}\ [\mathbf{cls}]\ )$$

The member variables have already been checked in Pass 2, they can be skipped in Pass 3. As the context as already been prepared by `top_III_cls` above, the type check of methods is the same as for procedures (Section 4.3.2):

$$\forall \begin{pmatrix} cls, acc, name, \\ name', formals, \\ fnames', ftys', \\ ret, ret', body, \\ body' \end{pmatrix} \frac{\begin{array}{l} \vdash \mathit{formals} \mapsto_f \mathit{fnames}' : \mathit{ftys}'\ \mathbf{export}^* \ldots \\ \vdash \mathit{ret} \mapsto_k \mathit{ret}' :_k * \\ + [\texttt{fwd\_formal\_to\_typing\_ref}]\ (\langle 1 : \texttt{formal} \rangle)\ ; \\ + [\text{return-type} : \mathit{ret}'] \vdash \mathit{body} \mapsto_c \mathit{body}' \end{array}}{\begin{array}{l} \mathbf{method}_{III}^{cls, acc}\ \mathit{name}/\mathit{name}_{cls}\ \mathit{cls}(\mathit{fnames}' : \\ \mathit{ftys}') : \mathit{ret}/\mathit{ret}'\mathbf{begin}\ \mathit{body}/\mathit{body}'\ \mathbf{end} \end{array}} \ (\texttt{method\_III'})$$

Constructors deviate from procedures in that the constructors of super classes are accessible. (See *base-class initializers* in C^{++} [Str97].) Following Java, calls to these constructors are written as function applications

$$\texttt{super}(\mathit{args})$$

thus it is sufficient to introduce the rule

$$\forall\left(\scriptstyle name',ftys,acc\right)\ \frac{\vdash \textbf{in cls}(acc) :\ \textbf{constructor}_{\textbf{cls}} \mapsto name' : (ftys) \rightarrow \textbf{cls}}{\textbf{super} \mapsto name' : (ftys) \rightarrow \textbf{cls}}\ (\texttt{super\_ctor}[\textbf{cls}])$$

into the context for checking the constructor body.

4.4.6 Field Access

The usual syntax

$$x.f \qquad x.f(args)$$

accesses field f in object x. The second notation integrates method calls smoothly if $x.f$ returns a function type. The check of $x.f$ is accomplished by rule `member`:

$$\forall\left(\substack{e,m,e',m',t,c,\\ acc}\right)\ \frac{\begin{array}{l}\vdash e \mapsto e' : \texttt{tid}[c]\\ \vdash \textbf{access}\ c\ \textbf{as}\ acc\\ \vdash \textbf{in}\ c(acc) :\ m \mapsto m' : t\end{array}}{e.m \mapsto e'.m' : t}\ (\texttt{member})$$

It first ensures that e has a non-constructed type, then checks the current access rights for that type and finally delivers the typing for the named field by matching against a field declaration in the context.

As derived classes contain all the fields of their super classes (possibly overriding some method bodies), the search a declaration in the `member` rule may not stop at the class c: The desired field may also be found in one of c's super classes. This iterative search is implemented by the following two rules:[8]

$$\forall\left(\scriptstyle cls,acc,p\right)\ \frac{\vdash \textbf{in}\ cls(acc) :\ p}{\textbf{in}\ cls(acc) :\ p}\ (\texttt{inherit\_base})$$

$$\forall\left(\scriptstyle cls,super,acc,p\right)\ \frac{\begin{array}{l}\vdash cls\ \textbf{inherits}\ super\\ \vdash \textbf{in}\ super(acc) :\ p\end{array}}{\textbf{in}\ cls(acc) :\ p}\ (\texttt{inherit\_step})$$

Like conversions (Section 4.2.7), these rules lead to non-termination if applied freely. A simple proof-grammar is used to restrict applicability:

$$\begin{array}{rcccl}\texttt{inherit\_base} & : & icl & \longrightarrow & *\\ \texttt{inherit\_step} & : & icl & \longrightarrow & *,\ icl\\ \texttt{member} & : & * & \longrightarrow & sub,\ *,\ icl\\ \texttt{open\_this} & : & * & \longrightarrow & *,\ icl\end{array}$$

[8]Using a Prolog `!` (cut) operator would stop the search at the first found declaration.

4.4.7 Super Class Conversion and Downcast

As a general principle, a value of a derived class may be used wherever a value of its super class is expected. Because conversions for automatic dereferences (Section 4.2.7) have to be introduced anyway, it is straightforward to integrate super class conversions:

$$\forall\left({\scriptstyle e,e',cls,super}\right)\ \frac{\vdash \mathit{cls}\ \textbf{inherits}\ \mathit{super}}{e : \texttt{tid}[\mathit{cls}] \leq_c e : \texttt{tid}[\mathit{super}]}\ (\texttt{super\_conv})$$

Conversions in Class are transitive by the rules

$$\forall\left({\scriptstyle e,e',e'',s,t}\right)\ \frac{\begin{array}{c}\vdash e \mapsto e' : s\\ \vdash e' : s \leq_c e'' : t\end{array}}{e \mapsto e'' : t}\ (\texttt{subsume})$$

$$\forall\left({\scriptstyle e,e',e'',t,t',t''}\right)\ \frac{\begin{array}{c}\vdash e : t \leq_c e' : t'\\ \vdash e' : t' \leq_c e'' : t''\end{array}}{e : t \leq_c e'' : t''}\ (\texttt{conv\_trans})$$

$$\forall\left({\scriptstyle e,t}\right)\ e : t \leq_c e : t\ (\texttt{conv\_refl})$$

The following proof grammar guides the application of the rules such that a linear chain of conversions is created (cf. also Section 4.3.3):

`super_conv`	:	*conv1*	$\longrightarrow$	$*$
`deref_conv`	:	*conv1*	$\longrightarrow$	
`conv_refl`	:	*conv*	$\longrightarrow$	
`conv_trans`	:	*conv*	$\longrightarrow$	*conv1*, *conv*

Furthermore, super class conversions can be restricted to the following rules and premises:

`apply_branch`	:	$*$	$\longrightarrow$	*sub*, $*$
`return`	:	$*$	$\longrightarrow$	$*$, *sub*
`new_op`	:	$*$	$\longrightarrow$	$*$, *sub*
`subsume`	:	*sub*	$\longrightarrow$	$*$, *conv*

The automatic conversion to super classes is complemented by a manual *downcast* facility

$$e\texttt{:>}c$$

which asserts that expression e yields an object of class c, even though the type checker may be only able to infer a less precise super class of c. A prototypical example is expression

```
i := cell.get() :> integer;
```

in Figure 4.4, where `cell` class asserts that the `get()` method returns an `object`, while the programmer is certain that an `integer` will be found.

Downcasts are handled by the following rule; the predicate `inherits_trans` calls for a recursive search through all of the super classes, it is implemented by a straightforward pair of rules `inherits_trans_base` and `inherits_trans_step`.

$$\forall\left({\scriptstyle e,cls,e',super}\right)\ \frac{\begin{array}{c}\vdash e \mapsto e' : \texttt{tid}[\mathit{super}]\\ \vdash \mathit{cls}\textbf{inherits}^{*}\mathit{super}\end{array}}{e :> \texttt{tid}[\mathit{cls}] \mapsto e' :> \mathit{cls} : \mathit{cls}}\ (\texttt{cast\_exp})$$

4.5 A Language for Generic Programming

Generic programming seeks to implement algorithms under minimal requirements on their input data types [MS89, MS94, MS96, Aus98, Sch96a, GSL97, MSSL99, Gas01].[9] I present the design and implementation of a language SAGA[10] that supports an STL-style [MS96, Aus98] formulation of generic algorithms in such a way that the requirements of algorithms are part of the algorithms' interfaces. When a generic algorithm A calls another generic algorithm A', then it is checked that the requirements of A entail the requirements of A'. Hence, all instantiations of A may legally call algorithm A'. This is in sharp contrast with the re-compilation semantics of the C^{++} template mechanism, where errors are detected only during the generation of instances [SL00]. When SAGA is translated to C^{++}, for example using [Wei03], all instantiations are guaranteed to succeed.

Work by C. Schwarzweller [Sch97, Sch02, Sch03] on verification of generic algorithms using the MIZAR proof checker suggests that it is desirable to express separately the required operations on parameter types and their expected behaviour: The operations constitute the *signature* of an algorithm, while their expected behaviour is captured by *adjectives*, which are symbolic names for formulae in a suitable logic. This setup allows a fine-grained description of an algorithm's requirements, in particular the requirements can vary easily between algorithms.

The work presented in this section integrates this insight into the design of a programming language and shows that the resulting language can be effectively type-checked using TCG. Schwarzweller [Sch02] has given a basic calculus for reasoning with adjectives, which has to be extended to be used in a type checker. Foremost, the instantiation of algorithms, concerning both parameter types and operations on theses types, cannot be computed in Schwarzweller's calculus. For instance, consider the simple algorithm `swap` [Aus98, Section 12.2].[11]

```
algorithm swap
[ (T, ‘= : (&T,T)->void ) with assignable(T,‘=) ]
( x : &T; y : &T)
begin
 var tmp : T;
 tmp = x;
 x   = y;
 y   = tmp
end
```

Even in the trivial call `var i,j : int; swap(i,j)` the compiler must determine that

1. `int` is the parameter type `T`
2. the built-in assignment on integers instantiates the operation `=`.

[9]A different reading of the term "generic programming" is concerned with recursion patterns over algebraic data types in functional programs. The intention is to abstract over the constructors of data types. (see for example [JJ97]). I do not pursue this interpretation.

[10]The name is due to Christoph Schwarzweller ("*S*ignatures and *A*djectives for *G*eneric *P*rogramming").

[11]The algorithm's signature requires one parameter type `T` and one operator `=`, which is written `‘=` since it is not in its usual infix position. The `with` clause introduces the requirement that `=` is assignment on `T`.

To determine these, the compiler generates a proof obligation

$$\{P\} \Longrightarrow \{\texttt{assignable}(T,=)\}$$

where P are the adjectives valid in the context of the call. For the type `int`, we have a declaration

```
assert assignable(int,=)
```

which by overload resolution for `=` gets translated to

$$\texttt{assignable}(\texttt{\_\_int}, \texttt{\_\_int\_assign}) \in P$$

Hence, the compiler can deduce that the call is valid with

$$\{T \mapsto \texttt{\_\_int}, \texttt{=} \mapsto \texttt{int\_assign}\} \; .$$

The calculus in [Sch02, Section 3][Sch03, Section 6.2] (see also Section 4.5.1) does not allow this instantiation to be performed. Using the rule mechanism of Tcg, a corresponding extension is natural and straightforward.

A second detail that has been contributed by the design presented in this section is the ellipsis construct (Section 4.5.3.5): Since complete specifications of signatures easily become unwieldy, the type checker can transfer unspecified signature elements, denoted by "...", from the definitions of adjectives. In the `swap` example, it is sufficient to specify

```
(T,...) with assignable(T,...)
```

and the type checker infers that the assignment operation is missing; it chooses the name `=` from the adjective definition.

A preliminary presentation of the Saga language itself, aside from the above considerations, is contained in [GS02]. An survey of the material has also appeared in [Sch03, Sections 6.2 and 6.3].

Overview This section is divided in three parts: Section 4.5.1 reviews the calculus of signatures and adjectives as presented by Schwarzweller [Sch02]. Section 4.5.2 introduces the basic programming language, which can be assembled from fragments in the library (Section 4.2). Section 4.5.3 introduces the features necessary for generic programming, which comprises signatures, adjective definitions, adjective applications, facts (assertions) and algorithms. Section 4.5.4 sketches how the result of the type check can be translated to C^{++}, such that the C^{++} compiler generates (without failures) the template instances. Section 4.5.5 applies Saga to a quick sort algorithm with STL iterators using the presentation of [Aus98].

4.5.1 The Calculus of Adjectives and Signatures

Schwarzweller [Sch02] gives a characterization of of theorems as triples

$$T = (\mathit{Sig}(T), \mathit{Cont}(T), \mathit{Prop}(T))$$

Here *Cont*(T) is the *content* of the theorem, i.e. the main statement; *Sig*(T) is the *signature*, which contains the constants and operator symbols appearing in the content, together with their arities (types). Finally, *Prop*(T) is a set properties that elements of the signature must possess in order for *Cont*(T) to be valid. These properties are expressed as symbolic adjectives, resembling the way that predicates are interpreted as relations in first-order logic. For example [Sch02, Section 2], the theorem

Let R be a (commutative) ring. Then $\{0\}$ *is an ideal in* R

would be expressed as a triple

$$\begin{aligned}&((R,+,*,0),\ \text{“}\{0\} \text{ is an ideal in } R\text{”},\\ &\quad \{\text{associative}(R,+),\ \text{right-zero}(R,+0),\ \text{right-inverse}(R,+,0),\ \text{distributive}(R,+,*)\})\end{aligned}$$

where the content would be expanded into a first-order formula. A *domain* D can likewise be expressed as a pair of a signature and the properties that a domain fulfills.

$$D = (\mathit{Sig}(D), \mathit{Prop}(D))$$

As an example, the integers may be described by

$$\begin{aligned}\mathit{Sig}(\mathbb{Z}) &\supseteq \{\mathbb{Z}, +_\mathbb{Z}, *_\mathbb{Z}, 0_\mathbb{Z}, 1_\mathbb{Z}\}\\ \mathit{Prop}(\mathbb{Z}) &\supseteq \left\{\begin{array}{l}\text{associative}(\mathbb{Z},+_\mathbb{Z}),\ \text{commutative}(\mathbb{Z},+_\mathbb{Z}),\ \text{commutative}(\mathbb{Z},*_\mathbb{Z}),\\ \text{distributive}(\mathbb{Z},+_\mathbb{Z},*_\mathbb{Z}), \text{Euclidean}(\mathbb{Z},+_\mathbb{Z},*_\mathbb{Z},0_\mathbb{Z})\end{array}\right\}\end{aligned}$$

Adjectives are names for properties. Hence, relationships between properties can be expressed at the symbolic level by *rules*

$$P \longrightarrow Q$$

where P and Q are sets of adjectives. Rules are extended to an *implication relation* $\Longrightarrow$ between sets of adjectives by the rules in figure 4.5. Schwarzweller defines its meaning as:

$$\models A_1 \Longrightarrow A_2 :\Longleftrightarrow\ \ \forall D : D \models A_1 \text{ implies } D \models A_2$$

Schwarzweller gives three rules **(B1–3)** to reduce a goal $P_1 \Longrightarrow P_2$ to the trivial goal $P_1 \Longrightarrow \varnothing$, which holds by **(AX1)**. The rules are [Sch02, section 3]:

(B1) A goal $\vdash P_1 \Longrightarrow P_2$ can be replaced by the goal $\vdash P_1 \Longrightarrow P_2 \setminus (P_1 \cap P_2)$.
(B2) A goal $\vdash P_1 \Longrightarrow P_2$ such that there is a rule $l \longrightarrow r \in L$ and a substitution σ with $\sigma(r) \cap P_2 \neq \varnothing$ can be replaced by the goal $\vdash P_1 \Longrightarrow (P_2 \setminus (\sigma(r) \cap P_2)) \cup \sigma(l)$.
(B3) A goal $\vdash P_1 \Longrightarrow P_2$ with $P_2 = \varnothing$ holds trivially.

Schwarzweller claims without proof that these rules are correct for the calculus in Figure 4.5, i.e. if they reduce a given judgement $\vdash P_1 \Longrightarrow P_2$ to the trivial judgement $\vdash P_1 \Longrightarrow \varnothing$, then there is a deduction using the rules **(AX1/2)**,**(R1/2)**. Completeness of **(B1–3)** is not claimed.

$$\frac{P_2 \subseteq P_1}{\vdash P_1 \Longrightarrow P_2} \qquad \textbf{(AX1)}$$

$$\frac{l \longrightarrow r \in L}{\vdash \sigma(l) \Longrightarrow \sigma(r)} \qquad \textbf{(AX2)}$$

$$\frac{\vdash P_1 \Longrightarrow P_2,\ \vdash P_1 \Longrightarrow P_3}{\vdash P_1 \Longrightarrow P_2 \cup P_3} \qquad \textbf{(R1)}$$

$$\frac{\vdash P_1 \Longrightarrow P_2,\ \vdash P_2 \Longrightarrow P_3}{\vdash P_1 \Longrightarrow P_3} \qquad \textbf{(R2)}$$

Figure 4.5: Implication relation for properties

4.5.2 The Programming Language

Saga is an imperative language enhanced with abstraction of algorithms over signatures to support generic programming. We can therefore reuse the Tcg fragments developed in Sections 4.3, 4.2.4, 4.2.2, and 4.2.7.

4.5.2.1 The Base Language

The type system of the base language has been kept as simple as possible, since our main interest is in the extension with signatures and attributes. Unlike usual imperative languages, we provide a function-type constructor and higher-order functions. The type language is given by the following grammar.

```
    type ::= tid | 'int' | 'char' | 'string' | 'bool'
           | 'void'
           | 'array' '(' type ')'
           | '<' type ',' type '>'
           | 'cell' '(' type ')'
           | '&' type
           | '(' typelist? ')' '→' typed
           | type ( '(' typelist ')' )?
           | '(' type ')'
typelist ::= type ( ',' type )*
```

The terminal **tid** allows identifiers starting with upper-case letters, the remaining types are standard. (`< >` denotes pairs and `&` denotes references in the C^{++} sense of mutable locations.[12] As in Section 4.3.3, dereferencing is automatic. We use kinds (Section 4.2.3) to check the consistency of type expressions. Currently, Saga does not feature subtyping. Overloading is permitted only in a form similar to that of the original Ada proposal: All possible type-correct interpretations of an expression are enumerated. If there is more than

[12]Note that because Saga does not include type inference, `&` does not incur the well-known inconsistencies [Tof90] that a naive combination of references with polymorphism may introduce.

one interpretation for an expression, the expression is considered illegal. (See Section 4.3.4 for a discussion.) Type checking of Saga algorithms breaks down the statements to single expressions (Section 4.2.4) and types these using the standard rules (Section 4.2.2). The reference to algorithm parameters, variables and the return type is analogous to Section 4.3. An algorithm's parameters are not considered mutable to facilitate re-use of library rules.

4.5.2.2 Type Definitions

Type definitions serve to introduce new types and type constructors.

$$\textit{type-def} \quad ::= \quad \texttt{'type'}\ \textbf{tid}\ \big(\ \texttt{'('}\ \textbf{tid}\ (\texttt{','}\ \textbf{tid})^*\ \texttt{')'}\ \big)^?\ \texttt{'='}\ \textit{type}$$

A type definition $\texttt{type}\ T(s_1 \mathinner{..} s_n) = t$ is legal iff t is a well-formed type assuming that s_i are types. These requirements are directly reflected by the following Tcg rule. The fresh name *name′* is chosen to resolve later references uniquely.

$$\forall \begin{pmatrix} \scriptstyle name, name', \\ \scriptstyle args, def, def', \\ \scriptstyle k, args', kinds' \end{pmatrix} \frac{\begin{array}{l} \vdash \text{extract arg } \mathit{args} \mapsto \mathit{args'} :: \mathit{kinds'}\ \textbf{export}^* \ldots \\ + \langle 1 : \texttt{tname} \rangle \vdash \mathit{def} \mapsto_k \mathit{def'} :_k * \\ \vdash \mathit{name} \overset{\text{newopq}}{\mapsto} \mathit{name'} \\ \vdash \textbf{def}\ \mathit{name}/\mathit{name'}(\mathit{args'}) = \mathit{def'}\ \textbf{export}\ \texttt{tydef!} \end{array}}{\textbf{type}\ \mathit{name}(\mathit{args}) = \mathit{def}} \ (\texttt{tydef})$$

In the current implementation, we do not have recursive types, but these are a simple technical extension as in the case of algorithms (see Sections 4.5.3.8, 4.1.3.6). Similarly the extension to higher-order type constructors is straightforward, using standard kind-inference techniques (e.g. [Jon95, Section 5]).

Type definitions are unfolded directly during the validity check for type expressions. Since by the above rule for type definitions, newly defined type constants must be applied to zero arguments, type application is the only place that defined types appear.[13]

$$\forall \begin{pmatrix} \scriptstyle c, c', args, args', \\ \scriptstyle res, k \end{pmatrix} \frac{\begin{array}{l} \vdash c(\mathit{args}) \mapsto_k c'(\mathit{args'}) :_k k \\ \vdash (c', \mathit{args'}) \overset{\text{unfold}^?}{\mapsto} \mathit{res} \end{array}}{c(\mathit{args}) \mapsto_k \mathit{res} :_k k} \ (\texttt{t\_app\_kind\_unfold})$$

Obviously, this rule introduces an infinite loop into the search, which we break by a local proof grammar. That grammar states that a single lookup will be sufficient at each application.

$$\begin{array}{rclcl} \texttt{t\_app\_kind} & : & \mathit{ty\_app\_no\_unfold} & \longrightarrow & *, * \\ \texttt{t\_app\_kind\_unfold} & : & * & \longrightarrow & \mathit{ty\_app\_no\_unfold} \end{array}$$

The unfolding predicate in `t_app_kind_unfold` needs two rules, one for the case that the constructor c is defined, and a default case which just constructs an application. Note

[13]It would be possible to insert the empty argument list automatically for defined types by checking for $() \to *$ in every lookup of a $\texttt{tid}[t]$. However, the typing rules from the library do not anticipate this indirection.

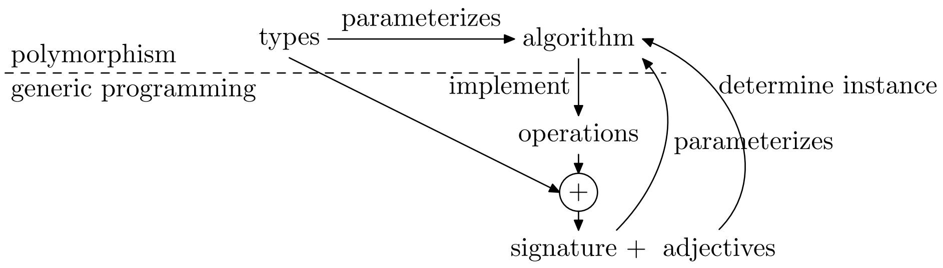


Figure 4.6: Relationship between algorithms and data types

that in the first case, the definition will have been unfolded already during the validity check of the type definition, such that it does not need processing now. In both cases, the arguments have already been unfolded in `t_app_kind_unfold`. (! is the cut operator.)

$$\forall\,(c, args, def)\ \frac{\vdash c(args) =_{\mathbf{def}} def \qquad !}{(c, args) \overset{\text{unfold}^?}{\mapsto} def}\ (\texttt{t\_app\_unfold\_def})$$

$$\forall\,(c, args)\ (c, args) \overset{\text{unfold}^?}{\mapsto} c(args)\ (\texttt{t\_app\_unfold\_undef})$$

4.5.3 Generic Programming

Generic programming Saga focuses on algorithms [MS94]. Instantiating generic algorithms replaces their type- and operation parameters with concrete types and operations determined at the call-site. (When generic algorithms call generic algorithms, these concrete arguments may again refer to algorithm parameters.) The parameters themselves are contained in an algorithm's *signature*, which is complemented by *adjectives* restricting the possible replacements (Section 4.5.1). On the other hand, generic algorithms can also be used as operations on data types themselves, hence the overall picture of Figure 4.6 arises. Besides restricting the valid instances of algorithms, adjectives in Saga, unlike in the underlying calculus [Sch02], play a second, important role in computing the instantiation of generic algorithms. Consider again the `swap` example from the introduction to this Section 4.5. The necessary instantiation is computed in two steps: First, unification of the actual and formal parameters' types yields $T =$ `__int`, the built-in integer format. Then, the type-checker seeks to prove

$$\texttt{assignable}(\texttt{\_\_int}, e)$$

where e is a variable. In the context of the call, we have a single declaration

$$\texttt{assignable}(\texttt{\_\_int}, \texttt{\_\_int\_assign})$$

and by unification, we get $e \mapsto$ `__int_assign`. (Compound expressions may also be used to instantiate e.)

At this point, the type checker must ensure that e is indeed bound to an expression of type $(\texttt{\&\_\_int}, \texttt{\_\_int}) \to \texttt{void}$. This is done indirectly: In the definition of the adjective `assignable` (Sections 4.5.3.3, 4.5.5), the types of its parameters are fixed, and every use of the adjective (Section 4.5.3.4) must respect that typing. For example, `assignable` has parameters

$$(\texttt{T}, \texttt{=} : (\texttt{\&T}, \texttt{T}) \to \texttt{void})$$

By unification of the arguments with the declared types the second argument must have type $(\texttt{\&\_\_int}, \texttt{\_\_int}) \to \texttt{void}$.

4.5.3.1 Signatures

The signatures and adjectives from Section 4.5.1 have a direct representation in Saga. A signature is a sequence of type names and operators with their arities.

$$\begin{aligned} \textit{signature} &::= \texttt{'('}\ \textit{sig-elem-list}\ \texttt{')'} \\ \textit{sig-elem-list} &::= \varepsilon \mid \textit{sig-elem}\ (\ \texttt{','}\ (\textit{sig-elem}))^* \\ \textit{sig-elem} &::= \textbf{tid} \mid \textbf{id}\ \texttt{':'}\ \textit{type} \end{aligned}$$

Signature elements are identified by their name and their type, hence operators may be overloaded within a signature. Note that besides operators, also constants can be *sig-elem*s. We will continue to speak of the *operators*, rather than the *values*, of the signature nevertheless, because this usage comes closer to the notion of algebraic data types.

4.5.1 Convention. For ease of notation, we assume in the subsequent description that the type elements of a signature precede the operator elements, i.e. signatures always have the form $(T_1 \mathinner{..} T_n, x_1 : t_1 \mathinner{..} x_m : t_m)$. The Saga language requires only that type name elements precede their uses in operations' types.

A signature $(T_1 \mathinner{..} T_n, x_1 : t_1 \mathinner{..} x_m : t_m)$ is legal if the types t_i of the operators are well-formed, under the assumption that $T_i :: *$ for all all $i \in 1 \mathinner{..} n$. The check is done by the following rules. They insert each found `tid` into the context of the subsequent checks. (See Section 2.5.2 for context modifiers in iteration premises.) The exports `vname` and `tname` will be referenced from outside the check to obtain the available types and operators. Hence, the check chooses fresh names to make these references unique.

$$\forall\left(\textit{elems},\textit{elems}'\right)\ \frac{\vdash \text{sign\_elem}(\textit{elems} \mapsto \textit{elems}')\ \textbf{export}^* \ldots + \langle 1 : \texttt{tname}\rangle}{\left(\textit{elems} \mapsto \textit{elems}'\right)}\ (\texttt{sign})$$

$$\forall\left(\textit{ty},\textit{ty}'\right)\ \frac{\begin{array}{l}\vdash \textit{ty} \overset{\text{newopq}}{\mapsto} \textit{ty}' \\ \vdash \textit{ty} \mapsto_k \textit{ty}' :_k *\ \textbf{export}\ \texttt{tname}!\end{array}}{\text{sign\_elem}((\texttt{tid}[\textit{ty}])_{\texttt{ty}} \mapsto (\textit{ty}/\textit{ty}')_{\texttt{ty}})}\ (\texttt{sign\_elem\_tid})$$

$$\forall\left(\textit{op},\textit{op}',\textit{ty},\textit{ty}',k\right)\ \frac{\begin{array}{l}\vdash \textit{op} \overset{\text{newopq}}{\mapsto} \textit{op}' \\ \vdash \textit{ty} \mapsto_k \textit{ty}' :_k k \\ \vdash \textit{op} \mapsto \textit{op}' : \textit{ty}'\ \textbf{export}\ \texttt{vname}!\end{array}}{\text{sign\_elem}((\textit{op} : \textit{ty})_{\texttt{op}} \mapsto (\textit{op}/\textit{op}' : \textit{ty}')_{\texttt{op}})}\ (\texttt{sign\_elem\_op})$$

4.5.3.2 Generalizing Signatures

Signatures act as parameters for generic algorithms. In algorithm calls, these parameters are variables to be instantiated. The following rules thus replace the newly chosen, unique names in a checked signature with TCG variables and record the choice in exports. We say that they *generalize* the signature.

$$\forall\left({}_{elems,elems'}\right)\ \frac{\begin{array}{c}\vdash \text{gen\_sign\_elem}(elems \mapsto elems') \\ \textbf{export}^{*}\ldots + \langle 1 : \texttt{tname}\rangle\end{array}}{\text{gen\_sign}(elems \mapsto elems')}\ (\texttt{gen\_sign})$$

$$\forall\left({}_{ty0,ty,ty'}\right)\ \frac{\vdash ty \mapsto_k ty' :_k * \ \textbf{export}\ \texttt{tname!}}{\text{gen\_sign\_elem}((ty0/ty)_{\texttt{ty}} \mapsto (ty')_{\texttt{ty}})}\ (\texttt{gen\_sign\_elem\_tid})$$

$$\forall\begin{pmatrix}{}_{op,op',op'',ty,} \\ {}_{ty',k}\end{pmatrix}\ \frac{\begin{array}{c}\vdash ty \mapsto_k ty' :_k k \\ \vdash op' \mapsto op'' : ty' \ \textbf{export}\ \texttt{vname!}\end{array}}{\text{gen\_sign\_elem}((op/op' : ty)_{\texttt{op}} \mapsto (op'' : ty')_{\texttt{op}})}\ (\texttt{gen\_sign\_elem\_op})$$

Suppose some construct C, for instance an algorithm header, in a SAGA program depends on signature S. To obtain a generalized, generic version of C, where all the references to S are replaced by TCG variables, four steps are necessary:

1. We check S and obtain a new signature S' with unique names for types and operators. The mapping from the names in S and the unique names is recorded in exports `tname` and `vname`.
2. We extend the context with the exports and check C, which yields a C', where all names referring to S are replaced by the corresponding unique names in S'.
3. We generalize the signature S' to S'', where the unique names are replaced by TCG variables. These are again recorded in exports `tname` and `vname`.
4. We extend the the context with these exports and re-check S' (with the same rules as Step 2). This effectively replaces the new fresh names with the new variables.

The intermediate versions S' and C' with unique names are introduced solely for overload resolution: Since the variables in the final version S'' of the signature will not provide any filtering, the generalization can take place only after all references have been made unique. Note also that the fresh variables can be quantified by rule extraction (Section 3.1.3), because they certainly appear only in the sub-proofs of the above four steps. This procedure will be used with type definitions (Section 4.5.2.2), adjective definitions (Section 4.5.3.3), assertions (Section 4.5.3.7) and algorithms (Section 4.5.3.8).
4.5.2 REMARK. Step 4 above is possible because the internal, compiled (in the sense of Section 4.2.1.1) form of types and expressions uses the same constructors as the abstract syntax tree. If th internal form differed from the input form, we would provide a new set of rules to traverse this representation.

One subtlety arises if C contains names from outer binding levels, for instance from the global environment. Then C' will contain the resolved names, which fail to be looked up in the second pass. Our solution is to explicitly modify the rules from the outer environment such that the internal names appear to be defined. This is accomplished by forward resolution with the following rule (and a corresponding rule for type names).

$$\forall\left({}_{x,x',t}\right)\ \frac{\vdash \texttt{id}[x] \mapsto x' : t}{x' \mapsto x' : t}\ (\texttt{fwd\_outer\_bound\_val})$$

This solution expresses the intention of the second pass constructively: The names from the signature are generalized, while the other names remain unmodified. A solution with a more operational flavor would be to leave untouched those identifiers that are not found in the second pass. This solution requires the *cut*, however.

4.5.3.3 Adjective Definition

Adjectives are used as symbolic names for properties of signature elements. To enable the type-check of operations (see the introduction to Section 4.5.3), they must be declared with the list of their parameters.

$$\begin{array}{rcl} \textit{adj-def} & ::= & \texttt{'adjective'}\ \mathbf{id}\ \textit{signature meaning} \\ \textit{meaning} & ::= & \texttt{'means'}\ \textit{formula} \\ & | & \texttt{'informal'}\ (\ \mathbf{StringLit} \mid \mathbf{tid} \mid \mathbf{id}\ )^{+} \end{array}$$

An adjective definition introduces **id** as a name of an adjective with *formula* or the textual description as its meaning. The types and operators from the *signature* may appear in the meaning clause. The precise shape of *formula* depends on the logic used for verification. No choice has been made as yet, and only the *informal* meaning is available. The second phrase for *meaning* can be used to give a textual description if program verification is not intended. For instance, the STL concepts in Section 4.5.5 are defined informally by references to Austern's collection [Aus98].

An adjective definition is valid if the signature is valid and the meaning is defined in a context enriched with the signature elements. Since in an informal meaning no overload resolution is possible, any previously declared operators overloaded in the signature are removed. This accounts for the first four premises.

$$\forall\begin{pmatrix}\scriptstyle name,sign, \\ \scriptstyle meaning,sign', \\ \scriptstyle sign'', \\ \scriptstyle meaning', \\ \scriptstyle meaning'', \\ \scriptstyle formals\end{pmatrix} \frac{\begin{array}{l}\vdash [\text{branch}:\textit{unique}]\big(\textit{sign} \mapsto \textit{sign}'\big)\ \mathbf{export}^{*} \\ \mathbf{expand} \\ -(\texttt{=>}_1\langle \textit{cls} = \texttt{id}\rangle);+\langle 1:\texttt{tname}\rangle;+\langle 1:\texttt{vname}\rangle \\ \ \vdash \text{informal}(\textit{meaning} \mapsto \textit{meaning}')\ldots \\ \vdash \text{gen\_sign}(\textit{sign}' \mapsto \textit{sign}'')\ \mathbf{export}^{*} \\ -(\texttt{=>}_1\langle \textit{cls} = \texttt{id}\rangle);+\langle 4:\texttt{tname}\rangle;+\langle 4:\texttt{vname}\rangle \\ \ \vdash \text{informal}(\textit{meaning}' \mapsto \textit{meaning}'')\ldots \\ \vdash \text{sig\_elem} \rightarrow \text{adjarg}(\textit{sign},\textit{sign}'') \mapsto \textit{formals}\ldots \\ \vdash \mathbf{def}\ \textit{name}(\textit{formals}) \equiv \textit{meaning}''\ \mathbf{export}\ \texttt{adjdef!}\end{array}}{\mathbf{adjective}\ \textit{name}\ \mathbf{for}\ \big(\textit{sign}\big)\ \mathbf{informal}\ \textit{meaning}}\ (\texttt{adjdef})$$

After the validity of the definition has been established, it must be made available in the context. To that end, the given signature is generalized and the already checked meaning is checked again under the generalized environment (see Section 4.5.3.2). The result is exported under name `adjdef` and can be added to subsequent contexts via the following forward rule:

$$\forall\begin{pmatrix}\scriptstyle adj,args, \\ \scriptstyle meaning\end{pmatrix} \frac{\vdash \mathbf{def}\ \textit{adj}(\textit{args}) \equiv \textit{meaning}}{\textit{adj} :_{\texttt{a}} \textit{args}}\ (\texttt{exp\_adjdef})$$

The exported *meaning* is not currently used. It is intended for a generation of proof obligations for facts (Section 4.5.3.7, Remark 4.5.4).

4.5.3.4 Adjective Application

An *adjective application* refers to a defined adjective by name, for instance to include its defined meaning as a requirement of a generic algorithm. The adjective's parameters are instantiated with types and expressions:

$$\textit{adj-apply} = \textbf{id}\ \texttt{'('}\ (\ \textit{type} \mid \textit{expr}\ )^*\ \texttt{')'}$$

An adjective application is valid if the types match type names in the adjective's definition and the expressions yields values of the types required by the signature's operators. The following rules capture these requirements.

$$\forall \begin{pmatrix} \textit{name},\textit{actuals}, \\ \textit{actuals}', \\ \textit{formals} \end{pmatrix} \frac{\begin{array}{c}\vdash \textit{name} :_{\texttt{a}} \textit{formals} \\ \vdash \texttt{adj\_arg}\ \big(\textit{actuals}, \textit{formals}\big) \mapsto \textit{actuals}' \ldots\end{array}}{\text{adj } \textit{name}(\textit{actuals}) \mapsto \textit{name}(\textit{actuals}')} \ (\texttt{chkadj})$$

$$\forall \begin{pmatrix} e,op,ty,e', \\ \textit{name} \end{pmatrix} \frac{\vdash e \mapsto e' : ty}{\texttt{adj\_arg}\ \big((e)_{\texttt{exp}}, (\textit{name} = e' : ty)_{\texttt{op}}\big) \mapsto (e')_{\texttt{exp}}} \ (\texttt{adj\_arg\_op})$$

$$\forall \big(ty,ty',\textit{tname}\big) \frac{\vdash ty \mapsto_k ty' :_k *}{\texttt{adj\_arg}\ \big((ty)_{\texttt{tyexp}}, (\textit{tname} = ty')_{\texttt{ty}}\big) \mapsto (ty')_{\texttt{tyexp}}} \ (\texttt{adj\_arg\_ty})$$

In these checks, the operators in the adjective instances can be overloaded. To that end, note that the generalized adjective definitions (Section 4.5.3.3) share the variables for the signature's types with the types of the signature's operators. For instance, writing

$$\texttt{assignable(int}, e)$$

already fixes the type of e to $(\texttt{\&int}, \texttt{int}) \to \texttt{void}$, and operators in e will be resolved correspondingly. For example, we can simply write

$$\texttt{assignable(int, `=)}$$

and the compiler will find out that = is the built-in integer assignment, as postulated in the introduction to this Section 4.5.3.
4.5.3 Remark. Because the type elements precede the operators using the types (Section 4.5.3.1), this overload resolution will always work on instantiated, ground types.

4.5.3.5 Expanding Signatures

The notions of signatures and attributes serve well to specify the interface of generic algorithm. Writing them down, however, is tedious. The language Saga therefore includes an *ellipsis* construct "..." that facilitates the most frequent case. If there is a pair of a signature $S = (s_1 \mathinner{..} s_n, \ldots)$ and an adjective application $a(p_1 \mathinner{..} p_m, \ldots)$, then the signature S is expanded such that it contains those parameters from a's definition that are not explicitly given by the $p_1 \mathinner{..} p_m$. The type and operator names from a's definition are chosen both for the signature and as new parameters to a. For example, the `assignable` adjective from the introduction would expand

$$(\ldots) \qquad \texttt{assignable}(\ldots)$$

into

$$(\texttt{T}, \texttt{=} : (\texttt{\&T}, \texttt{T}) \to \texttt{T}) \qquad \texttt{assignable}(\texttt{T}, \texttt{=})$$

Similarly for

$$(\texttt{S}, \ldots) \qquad \texttt{assignable}(\texttt{S}, \ldots)$$

$$(\texttt{S}, \texttt{=} : (\texttt{\&S}, \texttt{S}) \to \texttt{S}) \qquad \texttt{assignable}(\texttt{S}, \texttt{=}) \tag{4.5.1}$$

Expansion proceeds from left-to-right. It maps a triple

$$\langle \mathit{signature}, \mathit{actuals}, \mathit{formals} \rangle$$

to a pair of result signature and result actual parameters

$$\langle \mathit{signature}', \mathit{actuals}' \rangle$$

It thus replaces both ellipses in signature and adjective application with new elements. Towards that end, it unifies the *actuals* with the *formals* sequentially.

There are three cases to consider. When the *actuals* become empty, then the signature is not expanded.

$$\forall\, (\scriptstyle \mathit{sign}, \mathit{formals}) \ \text{expand}\ (\mathit{sign}, []) + \mathit{formals} \mapsto (\mathit{sign}, []) \ (\texttt{exp\_ellipsis\_base})$$

When an ellipsis is reached in the *actuals*, then the remaining *formals* are added to the signature.

$$\forall \begin{pmatrix} \scriptstyle \mathit{sign}, \mathit{sign}', \\ \scriptstyle \mathit{formals}, \mathit{args}' \end{pmatrix} \frac{\vdash \text{expand/merge}(\mathit{sign}, \mathit{formals}) \mapsto (\mathit{sign}', \mathit{args}')}{\text{expand}\ (\mathit{sign}, \ldots) + \mathit{formals} \mapsto (\mathit{sign}', \mathit{args}')} \ (\texttt{exp\_ellipsis\_ell})$$

If both an actual and a formal parameter are left, expansion unifies them. This unification instantiates the variables in types of operators following in the *formals*. (See for example (4.5.1), where `S` is transported to the type of `=` by this unification.)

$$\forall \begin{pmatrix} \scriptstyle \mathit{sign}, \mathit{sign}', a, \mathit{as}, \\ \scriptstyle \mathit{as}', f, \mathit{formals} \end{pmatrix} \frac{\begin{array}{l} \vdash \text{expand/unify}\ a \ = \ f \\ \vdash \text{expand}\ (\mathit{sign}, \mathit{as}) + \mathit{formals} \mapsto (\mathit{sign}', \mathit{as}') \end{array}}{\text{expand}\ (\mathit{sign}, a{::}\mathit{as}) + f{::}\mathit{formals} \mapsto (\mathit{sign}', a{::}\mathit{as}')} \ (\texttt{exp\_ellipsis\_step})$$

$$\forall\, (\scriptstyle \mathit{ty}, \mathit{tname}) \ \text{expand/unify}\ (\mathit{ty})_{\texttt{tyexp}} \ = \ (\mathit{tname} = \mathit{ty})_{\texttt{ty}} \ (\texttt{exp\_ellipsis\_unify\_ty})$$

$$\forall \begin{pmatrix} \scriptstyle e, \mathit{elem}, \mathit{name}, \\ \scriptstyle \mathit{ty} \end{pmatrix} \text{expand/unify}\ (e)_{\texttt{exp}} \ = \ (\mathit{name} = e : \mathit{ty})_{\texttt{op}} \ (\texttt{exp\_ellipsis\_unify\_exp})$$

The insertion of new signature elements is straightforward using the *cut* operator. We show only the match and base case.

$$\forall\,(\scriptstyle elem,sign\displaystyle)\ \frac{!}{\text{expand/ins}\,elem{::}sign + elem \mapsto elem{::}sign}\ (\texttt{exp\_ellipsis\_ins\_match})$$

$$\forall\,(\scriptstyle elem\displaystyle)\ \text{expand/ins}\ldots + elem \mapsto elem{::}\ldots\ (\texttt{exp\_ellipsis\_ins\_ellipsis})$$

When all expansions are complete, the final ... in the signature is replaced by `[]` straightforwardly.

4.5.3.6 Solving Proof Obligations

Proof obligations are represented by predicates **prove** and **prove***, where the latter one is immediately reduced to the former by the rule

$$\forall\,(\scriptstyle adjs\displaystyle)\ \frac{\vdash \mathbf{prove}(adjs)\ldots}{\mathbf{prove}^*(adjs)}\ (\texttt{prove\_all})$$

The judgement **prove**(A) represents a proof obligation A in the form of an adjective that must be valid in the current context. That context is again represented by Tcg's context, and resolution of proof obligations is currently done by Tcg's built-in search mechanism, that is SLD resolution with depth-first search. Note that Tcg's unification instantiates any remaining variables in proof obligation A, a capability that supersedes the calculus in Section 4.5.1.

4.5.3.7 Facts

Facts capture implications between sets of adjectives, as introduced in Section 4.5.1.

facts	::=	`'assert'` *fact*$^+$
fact	::=	`'for'` *signature rule*
rule	::=	`'{'` *adj-apply** `'}'` `'==>'` `'{'` *adj-apply*$^+$ `'}'`

If one of the sets of adjectives contains a single element, the braces may be omitted. If the left-hand-side set of adjectives is empty, the braces and the arrow may be omitted.

The facts of an `assert` clause are checked separately. A fact is valid if the contained adjectives are valid under the given signature. The following rule checks this requirement. Afterwards, it generalizes the fact according to Section 4.5.3.2 and exports it for later reference.

$$\forall\begin{pmatrix} {\scriptstyle sign, sign',} \\ {\scriptstyle sign'', sign''',} \\ {\scriptstyle sign'''', rl, lhs,} \\ {\scriptstyle lhs', lhs'', rhs,} \\ {\scriptstyle rhs', rhs''} \end{pmatrix} \frac{\begin{array}{l} \vdash \text{expand/rule } (sign, rl) \mapsto (sign', \{lhs\} \longrightarrow \{rhs\}) \\ \vdash \text{expand/close}(sign' \mapsto sign'') \\ \vdash \text{ground?}(sign'', lhs, rhs) \\ \vdash \big(sign'' \mapsto sign'''\big)\ \textbf{export}^* \\ +\langle 4 : \texttt{tname}\rangle;\ +\langle 4 : \texttt{vname}\rangle \\ \ \vdash \text{adjs } lhs \mapsto lhs' \text{ then } rhs \mapsto rhs' \\ \vdash \text{gen\_sign}(sign''' \mapsto sign'''')\ \textbf{export}^* \\ +[\texttt{fwd\_outer\_bound\_val}]\ (environment)\,; \\ \ +[\texttt{fwd\_outer\_bound\_ty}]\ (environment)\,; \\ \ +\langle 6 : \texttt{tname}\rangle;\ +\langle 6 : \texttt{vname}\rangle \\ \ \vdash \text{adjs } lhs' \mapsto lhs'' \text{ then } rhs' \mapsto rhs'' \\ \vdash \{lhs''\} \Longrightarrow \{rhs''\}\ \textbf{export}\ \texttt{factdecl!}\ldots[rhs''] \end{array}}{\forall\big(sign\big) : rl}\ (\texttt{fact})$$

The export of `factdecl` is by iteration through rhs'' with fixed lhs''. The exported implications thus have a singleton right-hand side and can be inserted to contexts as Tcg rules, where the left-hand side of the implication then becomes a set of proof-obligations (see Section 4.5.3.6). The following rule is used for forward resolution in rule extraction.

$$\forall\,({\scriptstyle l,r})\ \frac{\begin{array}{l} \vdash \{l\} \Longrightarrow \{r\} \\ \vdash \textbf{prove}^*(l) \end{array}}{\textbf{prove}(r)}\ (\texttt{exp\_factdecl})$$

In checking the adjective applications on the right-hand side we assume the adjectives on the left-hand side, which enables the use of (instances of) generic algorithms as operations. This construction parallels the natural deduction introduction of implication by discharging an assumption (Section 1.2.2.1). The following rule implements this dependency by another context modification.

$$\forall\begin{pmatrix} {\scriptstyle lhs, rhs, lhs',} \\ {\scriptstyle rhs'} \end{pmatrix} \frac{\begin{array}{l} \vdash \text{adj } lhs \mapsto lhs' \ldots \\ \vdash \textbf{prove}(lhs')\ \textbf{export}\ \texttt{assumption!}\ldots \\ \ +\langle 2 : \texttt{assumption}\rangle \vdash \text{adj } rhs \mapsto rhs' \ldots \end{array}}{\text{adjs } lhs \mapsto lhs' \text{ then } rhs \mapsto rhs'}\ (\texttt{chk\_rule\_adjs})$$

To continue the introductory example, we can then for example define

```
adjective swappable
for (T, swap : (&T,&T)->void)
informal swap " swaps values at argument locations"

assert for (T,...) assignable(T,...) ==> swappable(T,swap)
```

The the last line, we get a Tcg rule[14] instantiation of the `swap` algorithm with the required

[14]Transliterated from the raw output manually, TcgDoc currently does not handle run-time terms.

assignment from `assignable`.

$$\forall(T,e)\ \frac{\textbf{prove}(\texttt{assignable}(T,e))}{\textbf{prove}(\texttt{swappable}(T,\texttt{swap}\langle T,e\rangle))}$$

4.5.4 Remark. Semantically, a declared implication must be supported by the meanings of the given adjectives. Hence, there is an implicit proof obligation associated with a rule declaration: If for some instance of the signature, the meaning of the left-hand side adjectives are valid, then also the meanings of the right-hand side adjectives must be valid. These proof obligations could be generated by replacing the types and operators in the adjectives meaning clauses (Section 4.5.3.3) by the actual parameters.

4.5.3.8 Algorithm Definitions

Algorithm definitions in Saga have the form

$$\begin{array}{rcl}
\textit{algorithm} & ::= & \texttt{'algorithm'}\ \textbf{id}\ \textit{sig-params}^{?}\ \textit{val-params}^{?}\ \textit{return}^{?} \\
 & & \texttt{'begin'}\ \textit{statement-list}\ \texttt{'end'} \\
\textit{val-params} & ::= & \texttt{'('}\ (\ \textbf{id}\ \texttt{':'}\ \textit{type}\ (\ \texttt{';'}\ \textbf{id}\ \texttt{':'}\ \textit{type}\ )^{*}\ )^{?}\ \texttt{')'} \\
\textit{return} & ::= & (\ \texttt{'return'}\ \textit{type}\ )^{?} \\
\textit{sig-params} & ::= & \texttt{'['}\ (\ \textit{sig-param}\ )^{*}\ \texttt{']'} \\
\textit{sig-param} & ::= & \textit{signature}\ \texttt{'with'}\ \textit{adj-apply}\ (\texttt{','}\,\textit{adj-apply})^{*}
\end{array}$$

There are no forward declarations of algorithms. Mutually recursive groups of algorithms are defined by concatenation with the keyword **and**. Due to this recursion, each algorithm group is checked in two phases (see also Sections 4.4, 4.1.3.6): First, the algorithm headers are checked for consistency and generalized (Section 4.5.3.2). Then, they are inserted to the context for the check of the algorithm bodies.

The requirements for an algorithm definition to be valid are as follows:

$$\begin{array}{rl}
A = & \texttt{algorithm}\ a \\
 & \left[\left(S_i\ \texttt{with}\ (A_{ij})_{j=1}^{m_i}\right)_{i=1}^{n}\right] \\
 & (x_k : t_k)_{k=1}^{r} \\
 & \texttt{return}\ s \\
 & \texttt{begin}\ B\ \texttt{end}
\end{array}$$

- All signatures S_i are valid.
- All adjective applications A_{ij} are valid in a context enriched with signature S_i for every $i = 1 \mathinner{..} n$, $j = 1 \mathinner{..} m_i$.
- The *value-params* and *return* include only well-formed types, possibly referring to types from the signatures S_i.
- The *statement-list* can be type-checked in a context enriched with
  - the types and operators from all the S_i
  - the assumptions that A_{ij} hold.

Neither types nor operators are shared between signatures. When the same operator appears in different signatures, it is overloaded (in just the same way that it is overloaded when it appears in the same signature with different types).

$$\forall \begin{pmatrix} \mathit{name}, \mathit{uname}, \\ \mathit{uname}', \\ \mathit{signadj}, \\ \mathit{signadj}', \\ \mathit{signadj}'', \mathit{args}, \\ \mathit{args}', \mathit{args}'', \\ \mathit{params}, \mathit{adjs}, \\ \mathit{ret}, \mathit{ret}', \mathit{ret}'', \\ \mathit{body}, \mathit{body}' \end{pmatrix} \dfrac{\begin{array}{l} \vdash \text{sign+adj} : \ \mathit{signadj} \mapsto \mathit{signadj}' \ \textbf{export}^{*} \ldots \\ + \langle 1 : \texttt{tname} \rangle \vdash \text{algarg} \ \mathit{args} \mapsto \mathit{args}' \ldots \\ + \langle 1 : \texttt{tname} \rangle \vdash \mathit{ret} \mapsto_{k} \mathit{ret}' :_{k} * \\ \vdash \text{choose} \ \mathit{name}/\mathit{uname} \to \mathit{uname}' \\ \vdash \mathit{name}(= \mathit{uname}') \equiv \big[\mathit{signadj}'\big]\big(\mathit{args}'\big) \to \mathit{ret}' \\ \qquad\qquad \textbf{export} \ \texttt{alghead}! \\ \vdash \text{gen\_signadj}(\mathit{signadj}' \mapsto \mathit{signadj}'') \ \textbf{export}^{*} \ldots \\ + \langle 6 : \texttt{tname} \rangle ; + [\texttt{fwd\_outer\_bound\_ty}] \ (\mathit{environment}) \\ \qquad\qquad \vdash \text{extract\_type} \ \mathit{args}' \mapsto \mathit{args}'' \ldots \\ + \langle 6 : \texttt{tname} \rangle ; + [\texttt{fwd\_outer\_bound\_ty}] \ (\mathit{environment}) \\ \qquad\qquad\qquad \vdash \mathit{ret}' \mapsto_{k} \mathit{ret}'' :_{k} * \\ \vdash \text{split} \ \mathit{signadj}'' \ \text{sign} = \mathit{params}, \ \text{adjs} = \mathit{adjs} \\ \vdash \textbf{def} \ \mathit{name} = \mathit{uname}' \langle \mathit{params} \rangle : (\mathit{args}'') \to \\ \qquad \mathit{ret}'' \ \text{requires} \ \mathit{adjs} \ \textbf{export} \ \texttt{algdef}! \end{array}}{\begin{array}{l} \text{check alghead}\big(\textbf{algorithm} \ \mathit{name}(= \mathit{uname}) \equiv \\ \big[\text{signadj}^{*} \mathit{signadj}\big] \big(\mathit{args}\big) \to \mathit{ret}\,\mathit{body}\big) \mapsto \mathit{uname}' \end{array}} \ (\texttt{alghead})$$

Figure 4.7: Checking the Header of a Generic Algorithm

Header The first phase of the recursion checks the algorithm headers, replacing all identifiers with unique counterparts to identify the references. The rule `alghead` (Figure 4.7) thus enforces the above requirements except for the last one.

We inspect the rule's premises one-by-one. First, the signatures and adjectives are checked as described in Sections 4.5.3.1 and 4.5.3.4. With exported type names, the algorithm's parameters and return types must be valid, which is expressed by the next two premises. We then choose a fresh, unique name for the algorithm, which is also accessible in the rule's consequence for later reference. This finishes the validity check, and the result (with unique, internal names) is recorded in an export. This export will be referenced by the unique name *uname′* before the check of the body. The remaining premises generalize the signature, and subsequently the remainder of the algorithm head as described in Section 4.5.3.2. (The auxiliary predicate `gen_signadj` iterates over all signatures and adjectives in the header.) The final `split` judgement separates the signatures, which represent the algorithm's generic parameters, from the adjectives, which constrain the instantiation. That generalized interface is also exported for later reference. Note that in the instance $\mathit{uname}'\langle \mathit{params} \rangle$ all the instantiation information is bundled with the unique internal name. Whenever the algorithm is referenced, the compiled expression (Section 4.2.1.1) will contain the necessary instantiation explicitly.

The export `algdef` will be referenced via the following forward rule:

$$\forall \begin{pmatrix} \mathit{inst}, \mathit{ty}, \mathit{name}, \\ \mathit{adjs} \end{pmatrix} \dfrac{\begin{array}{l} \vdash \textbf{def} \ \mathit{name} = \mathit{inst} : \mathit{ty} \ \text{requires} \ \mathit{adjs} \\ \vdash \textbf{prove}^{*}(\mathit{adjs}) \end{array}}{\mathit{name} \mapsto \mathit{inst} : \mathit{ty}} \ (\texttt{exp\_algdef})$$

Note how the saved adjectives become proof obligations in the context of each call to the algorithm. The conclusion of the rule, however, resembles an ordinary declaration of identifier *name* with internal, compiled representation *inst* and type *ty*. Rule extraction with quantification replaces the type parameters in *inst*, *ty* and *adjs* with bound Tcg variables, such that the algorithm can be used with different instantiations at each call.

Group To show the link between the head and body checks, here is the rule for the algorithm groups. Note in particular how the newly chosen *unames* are handed on from the headers to the bodies, and how the `algdef` exports are extracted quantified to allow mutual references to the generic algorithms.[15]

$$\forall\begin{pmatrix} \mathit{algs}, \mathit{unames}, \\ \mathit{bodies}' \end{pmatrix} \quad \frac{\begin{array}{l} \vdash [\text{branch }{:}\mathit{unique}]\text{check alghead}\big(\mathit{algs}\big) \mapsto \\ \quad \mathit{unames}\ \mathbf{export}^* \ldots \\ \mathbf{expand} \\ \quad + [\texttt{exp\_algdef}]\,(\langle 1 : \texttt{algdef}[\forall]\rangle)\,;\ + \langle 1 : \texttt{alghead}\rangle \\ \quad \vdash [\text{branch }{:}\mathit{unique}]\text{body}(\mathit{unames}) : \mathit{algs} \mapsto_c \mathit{bodies}' \ldots \end{array}}{\mathbf{algorithms} : \mathit{algs}} \quad (\texttt{alggrp})$$

Body An algorithm's body is checked straightforwardly: Using the unique name *uname*, the first premise looks up the `algdef` export stored in the context by `alggrp` above. An auxiliary judgement turns the signatures, adjectives and value parameters to referenceable facts, and `algbody` proceeds to check the statements under these added facts.

$$\forall\begin{pmatrix} \mathit{head}, \mathit{body}, \\ \mathit{name}, \mathit{uname}, \\ \mathit{signadj}, \mathit{args}, \\ \mathit{ret}, \mathit{body}' \end{pmatrix} \quad \frac{\begin{array}{l} \vdash [\mathit{trace\ algbody}, \mathit{uname}] \\ \vdash \mathit{name}(= \mathit{uname}) \equiv \big[\mathit{signadj}\big]\big(\mathit{args}\big) \to \mathit{ret} \\ \vdash \text{extract decls from } \mathit{signadj}, \mathit{args}\ \mathbf{export}^* \\ \quad + \texttt{decl\_return}[\mathit{ret}];\ + \langle 3 : \texttt{headdecl}\rangle \vdash \mathit{body} \mapsto_c \mathit{body}' \end{array}}{\text{body}(\mathit{uname}) : \mathbf{begin}\ \mathit{body}\ \mathbf{end} \mapsto_c \mathit{body}'} \quad (\texttt{algbody})$$

It is this setup that lets the body assume that the adjectives stated in the head are indeed valid, thus enabling the local checks for well-definedness that are at the core of Saga's design.

4.5.4 Translation to C^{++}

When a Saga program has passed all validity checks, the meaning of all algorithm bodies has been determined by overload resolution and a list of required instances of generic algorithms can be generated. Furthermore, all variable- and parameter references are unique because of the fresh names chosen for all bound names. The internal form of expressions can thus be translated to C^{++} directly, such that the C^{++} compiler does the instance generation without further overload resolution. We sketch the translation here, it can be effected by an external representation (Section 3.1.5).

[15]The semi-unification problem of polymorphic recursion [KTU93, Hen93] does not arise, since we are not attempting type inference. Instead, the `algdef` exports are ground terms, with unique type *names* for the algorithms' parameters.

Parameters are in general represented by template-definitions. We use `class` parameters only and wrap up operators into types in the spirit of function objects [MS96, Section 2.4]. Every such wrapper class has a `static` method `call`.

Algorithms are mapped to template `class` definitions with a single static method `call`. We have avoided the use of template functions, because the C^{++} compiler might accidentally perform overload-resolution on function names, which is not the case for `class` names. The schema is

```
template < class T₁ .. class Tₙ, class op₁ .. class opₘ >
struct algorithm name {
  static rtype call(ty₁arg₁ .. tyᵣargᵣ) {
    algorithm body
  }
};
```

Algorithm calls are then explicitly instantiated references to algorithm structures: *algname*<*param*$_1$ .. *param*$_n$>`::call(`*arg*$_1$ .. *arg*$_m$`)`

Type definitions have only type parameters. We use the C^{++} template mechanism to map a type name with parameters to the representation type. This can be achieved by a definition

```
template < class param₁ .. class paramₙ >
struct type name {
  typedef representation type def;
};
```

Type references to defined types with given instantiation are compiled into a template expression
typename<*param*$_1$ .. *param*$_n$>`::def`

Expanding Type Definitions Since SAGA expands type expressions, the above considerations are needed only for files other than the one where the type is introduced; in that file, all references to the type name have been expanded.

Primitive Operations on built-in data types (Section 4.5.4.1) are provided in a header file adhering to the above conventions concerning operator calls. The built-in types are hand-crafted as regular template `class` definitions.

With this mapping, the C^{++} compiler does not need to resolve any overloading nor instantiation and hence serves only to generate the code for required instances. Furthermore, since all calls are static and the `call` method bodies are given in the definition, the optimizer will be able to inline them.

4.5.4.1 Run-time System

In the subsequent examples (Section 4.5.5) we assume the following minimal run-time system. It provides primitive built-in types for machine integers, characters and strings. Both integers and characters have their C-semantics, while strings are handled by the C^{++} `string` class. Integers provide the usual arithmetic operations; string concatenation is denoted by operator `+`. The type `&` is mapped to C^{++} pointers, the necessary dereferences are inserted automatically (Section 4.3.3).

Arrays `[`T`]` have reference semantics. There are four built-in operations on arrays, where `[]` is written infix as a`[`i`]` as usual.

$$\begin{aligned} \texttt{new\_array}\colon (\texttt{int}, T) &\to \texttt{Array}(T) & \texttt{length}\colon (\texttt{Array}(T)) &\to \texttt{int} \\ =\colon (\&\,\texttt{Array}(T), \texttt{Array}(T)) &\to \&\,\texttt{Array}(T) & \texttt{[]}\colon (\texttt{Array}(T), \texttt{int}) &\to \&T \end{aligned}$$

Pairs `<`S`,`T`>` provide construction and (reference) access to their components.

$$\begin{aligned} \texttt{mkpair}\colon (S, T) &\to \texttt{Pair}(S, T) \\ \texttt{fst}\colon \texttt{<}S\texttt{,}T\texttt{>} &\to \&S \\ \texttt{snd}\colon \texttt{<}S\texttt{,}T\texttt{>} &\to \&T \end{aligned}$$

4.5.5 Generic Sorting in Saga

The C^{++} *Standard Template Library* [MS96] is often cited as an outstanding example of generic algorithms and data structures. Its flexibility largely stems from the decoupling of algorithms and the data structures they work on; this goal is achieved by the notion of *iterators*. Iterators abstract over the concrete details of data access and replace them by general movement and dereferencing operators. They come in five categories [MS96, Chapter 4]: Forward, input, output, bidirectional, and random-access iterators. A forward-iterator is both an input and an output iterator, a random-access iterator is a bidirectional iterator, and every bidirectional iterator is a forward iterator. The iterator categories are characterized by the available operations and their semantics: Forward iterators only require `operator++` to be defined, bidirectional iterators add `operator--` and finally random access is available via operators `+=`, `-=` and `[]`. To facilitate reasoning about wide-spread collections of operations and requirements, Austern [Aus98] groups them into named *concepts*. (Concepts were present in [MS96], but the Austern's presentation treats them more rigorously as objects in their own right.) A type is a *model* of a concept, iff it fulfills all the requirements associated with the concept.

We will show how SAGA enables reasoning about generic algorithms and semantic checking in terms of concepts. The mapping is straightforward: Signatures capture required operations and adjectives describe the expected properties. We use the concept names as adjectives. Our example is a fragment of the generic algorithms for sorting as found in the SGI implementation [Aus98]. We restrict the presentation to the fundamental `swap`, `partition`, `median`, and `sort` (quick sort). For lack of a “best match” overload resolution, we cannot represent the efficiency considerations for special cases found in the C^{++} implementation. Furthermore, since SAGA does not currently provide a `const` modifier for references, we cannot model the overloaded versions of iterator dereferencing. (Overload resolution for `const` vs. non-`const` references requires a “best match” strategy again.)

However, we do not change the requirement specification, which we directly translate from [Aus98]. In adjective definitions, we give an `informal` description of the semantics.

4.5.5.1 Basic Concepts

In the STL [MS96, Aus98], the primitive operations `==` (equality) and `=` (assignment) are not taken for granted. Consistently with the design of C^{++} [Str97], it is assumed that the programmer will in general overload these operators. The following adjectives capture the existence of the operations.

```
adjective assignable
for (T, ‘= : (&T,T)->void )
informal "Operator " ‘= " is an assignment on " T

adjective equality_comparable
for (T, ‘== : (T,T)->bool, ‘!= : (T,T)->bool)
informal "Operator " ‘== " is structural comparison on " T

assert assignable(int,‘=)
assert equality_comparable(int,‘==,‘!=)
```

Note that unlike in C^{++}, the names of the operators are not fixed. If `T` is some type and the structural comparison on `T` is called `equal_struct_T`, then we could as well write (\ is lambda abstraction).

```
assert equality_comparible(T,equal_struct_T, (\x. ! equal_struct_T(x)))
```

4.5.5.2 Iterator Concepts

For the sorting example, we are interested in two iterator categories that arise in the requirements for the `partition` and `sort` algorithms. First, partition needs to traverse a given sequence forward from the beginning and backward from the end. It needs the following `bidirectional_iterator` as input. (Saga represents operator names outside of their infix position by a prefixed single quote. For pre- and postfix operators, the argument position is indicated with a dot.)

```
adjective bidirectional_iterator
for (T, VT,
     ‘=   : (&T,T)->void,
     ‘==  : (T,T)->bool,    ‘!=  : (T,T)->bool,
     ‘++. : (&T)->&T,       ‘.++ : (&T)-> T,
     ‘--. : (&T)->&T,       ‘.-- : (&T)-> T,
     ‘*.  : (T)->&VT)
informal T "is a bidirectional iterator, with "
         ‘.-- " as backward step"
```

Quick sort must access a given pivot element in the middle of the input sequence, hence it requires random access. (It is not necessary to add the new operations at the end of the adjective, this is only done by convention.)

```
adjective random_access_iterator
for (T, VT,
     ‘=   : (&T,T)->void,
     ‘==  : (T,T)->bool,   ‘!=  : (T,T)->bool,
     ‘++. : (&T)->&T,       ‘.++ : (&T)-> T,
     ‘--. : (&T)->&T,       ‘.-- : (&T)-> T,
     ‘*.  : (T)-> &VT,
     ‘+   : (T,int)->T,     ‘-   : (T,T)->int)
informal T "is a random access iterator, with "
         ‘+ " and " ‘- " for random access"
```

When quick sort calls partition, the type checker must deduce that its requirements are sufficient to use that algorithm. Therefore, the implication that every random access iterator is also a bidirectional iterator must be made explicit.

```
assert
 for (T,VT,...)
 random_access_iterator(T,VT,...) ==> bidirectional_iterator(T,VT,...)
```

4.5.5.3 Swap

The basic algorithm `swap` [Aus98, Section 12.2.1] has the C^{++} interface

```
template<class Assignable>
void swap(Assignable &a, Assignable &b);
```

where `Assignable` is a model of the concept assignable.

In SAGA, we use the above defined adjective `assignable` in the specification of algorithm `swap`. When this generic algorithm is instantiated, the operator `=` will be replaced with some function `f` on the instantiation type `T’`, such that the user has declared the relation `assignable(T’,f)`.

```
algorithm swap
[ (T,...) with assignable(T,...) ]
( x : &T; y : &T)
begin
 var tmp : T;
 tmp = x;
 x   = y;
 y   = tmp
end
```

4.5.5.4 Partition

The algorithm partition [Aus98, Section 12.8.1] has the interface

```
template<class BidirectionalIterator, class Predicate>
BidirectionalIterator partition(BidirectionalIterator first,
                                BidirectionalIterator last,
                                Predicate pred);
```

where `BidirectionalIterator` is a model of bidirectional_iterator, `Predicate` is a model of predicate (i.e. an object applicable to a value to return `bool`) and `BidirectionalIterator`'s value type is convertible to `Predicate`'s argument type. The second template parameter `Predicate` is a peculiarity resulting from the use of function objects [MS96, Section 2.4]. We use λ-abstraction instead. As SAGA does not currently feature subtypes, convertibility reduces to equality[16] and the algorithm's interface can be written down directly:

```
algorithm partition
[ (T,VT,...)
   with bidirectional_iterator(T,VT,...), assignable(VT,...) ]
(fst : T;
 lst : T;
 pred  : (VT)->bool)
return T
```

The implementation of `partition` copies the parameters (parameters currently are not variables) and repeatedly applies `operator++` to `first` and `operator--` to `last`, which explains the first requirement. The second requirement on the value type of `T`, i.e. the type returned by dereferencing an iterator of type `T`, justifies a call to `swap`.

```
  var first : T;
  var last  : T;
  first=fst; last=lst;
  while (first != last) do
  begin
    if (pred(* first))
    then ++first
    else if (pred(* last))
    then --last
    else swap(* first, * last)
  end;
  return fst
```

Note that the operators `--`, `++`, `!=`, `=`, `*` in this algorithm are resolved to refer to the generic parameters, which are introduced to the ellipsis by expansion (Section 4.5.3.5).

4.5.5.5 Quick-Sort

The algorithm `sort` [Aus98, Section 13.1.1] requires random-access iterators.

```
template<class RandomAcessIterator>
void sort(RandomAcessIterator first,
          RandomAcessIterator last);
```

[16]We could introduce an adjective `convertible`. However, proof obligations with this adjective take the form of constraints deferred to future work (Section 6.2.3).

The requirements are that `RandomAcessIterator` is a model of RandomAcessIterator, and `RandomAcessIterator`is mutable, and `RandomAcessIterator`'s value type is StrictWeaklyComparable, that is there is an operator `<` implementing a strict weak order [Aus98, Section 6.4.2].

In Saga, the algorithm header is therefore

```
algorithm qsort
[ (RAI,VT,...)
  with random_access_iterator(RAI,VT,...),
       equality_comparable(VT,...),
       strict_weak_order(VT,...),
       assignable(VT,...)
]
( first : RAI;
  last  : RAI )
```

Note how the requirements on `VT` can be accumulated in a natural fashion using adjectives.

The implementation uses the median-of-three heuristics for selecting the pivot element, calls `partition` and recursively sorts the parts. The requirements from the `qsort` header are needed to check the validity of both calls to `partition` and `sort`, since the requirements of generic algorithms are noted in their type assumptions (Section 4.5.3.8).

```
  if (first == last) then return;
  var pivot : VT;
  pivot = select_pivot(first,last);
  var split : RAI;
  split = partition(first, last, (\x. x < pivot));
  qsort(first,split);
  qsort(split,last)
```

Chapter 5

Related Work

5.1 Tools

Starting in the 1980s, research on formalisms for programming language descriptions has been prospering and continues to produce useful tools for generating compiler components and programming environments [HK00, BHKO02, HPM$^+$02]. The search is driven on the one hand by the desire to generate interactive, language-specific programming environments, on the other hand by the need to check the consistency of formal language specifications.

For work related with TCG, three basic approaches seem relevant: *Attribute grammars*, *context relations* and *inference rules*. We choose the original proponent systems *The Synthesizer Generator*, *PSG* and *CENTAUR* to discuss these formalisms. As the differences from TCG manifest themselves in the basic formalism, rather than the specific tool, this discussion implicitly comprises more recent work based on the same formalisms. A further development, the ASF+SDF formalism (Section 5.1.3), is based on conditional rewriting. It has also been used for specifying type checkers [Hen89a, Deu96, TD01].

A more extensive survey on language description tools, without our focus on type systems, has been given by Heering and Klint [HK00]. They discuss a number systems with their respective capabilities [HK00, Table2, Figure 4], on which we have partially based our selection. A later overview of references is given by Henriques et al. [HPM$^+$02].

A recent contribution by Pierce and Levin [LP03] will be surveyed separately in Section 5.1.5. The developments based on constraint solving (e.g. [SS01, Sul01]) are covered in Section 5.2.

5.1.1 The Synthesizer Generator

The *Synthesizer Generator* [RT88] provides language-specific editors from language descriptions. It builds on previous work by the authors [TR81, Rep84] and relies on the formalism of *attribute grammars* [Knu68] for expressing the context-sensitive aspects of a language grammar. The generated editors are to incorporate knowledge about the (static) semantics of the edited language such that immediate feedback on errors can be given to the programmer. The scope of the treatment comprises imperative languages with finite, monomorphic types (Pascal, Ada, Modula). A restricted form of type inference [RT88, Section 7.2] is introduced to check for the inconsistent use of variables in a Pascal dialect.

An attribute grammar [RT88, Chapter 3] is a context-free grammar where each non-terminal has a set of associated *attributes* and each production has a set of *attribute equations*. Whenever a production is applied, the constructed abstract syntax tree is decorated with *attribute occurrences* of these attributes. Attribute equations then constitute side-conditions for the production to be applicable. An equation $A = e$ defines attribute occurrence A by attribute expression e, which references other attribute occurrences of the production. If A occurs on the left-hand-side of the grammar production it is *synthesized*, while the occurrences on the right-hand side are *inherited*. Any attribute of a given non-terminal must either be synthesized or inherited, i.e. an attribute cannot have a defined occurrence both on the left and the right-hand side of a production. In a *well-formed* grammar, the start production must not have inherited attributes and each attribute occurrence in a production must be defined exactly once. A grammar is *non-circular* if there are no derivation trees with cyclic attribute definitions.

Attribute expressions can be evaluated greedily [RT88, Section 12.1] (or *data driven*): Each node in the abstract syntax tree is labeled both with the non-terminal of the deriving production and the attributes of that non-terminal. As all of the production's attribute equations are local to the node (i.e. they are defined in the production itself or in its immediate neighbors), a simple inspection of these attributes determines when a given attribute equation can be evaluated, yielding a newly defined attribute that may again trigger evaluations. Furthermore, the dependencies between attributes indicate when attribute values become *inconsistent*, that is they do not satisfy the attribute equations. The detection of inconsistent attribute values then triggers computations to update these attribute values. The remainder of [RT88, Chapter 12] is dedicated to the pre-computation of evaluation strategies by static analysis of the grammar to increase the efficiency.

The flow of information in attribute grammars is inherently directional, somewhat contradictory to the intention of declarative descriptions for the language's semantics. Specifically, the information flow in classical Hindley-Milner [Mil78] type inference cannot be modeled straightforwardly, since here the type information is propagated by symmetric unification. Reps and Teitelbaum give a limited form of type inference where the types of variables are derived from their uses [RT88, Section 12.2]. They work in the finite flat lattice with types `Empty` (top), `NoType` (bottom), `Int` and `Bool`. To gather information on variable uses through synthesized attributes, they compute the meet of inferred types. They observe that assignments between variables introduce equality constraints (in a setting without coercions) on the types of variables without further narrowing their possible types. It appears that ML inference can be treated following Wand's [Wan87] presentation, if the meet computation is replaced by native unification; this approach is, however, not supported specifically by the underlying formalism.

5.1.2 The Programming System Generator (PSG)

The PSG system by Bahlke and Snelting [BS86] is a generator for language-specific programming environments. Like the synthesizer generator, its main emphasis is on interactive and incremental static analysis of incomplete program fragments. Its formalism likewise assigns attributes to the nodes of abstract syntax trees, which due to incompleteness of the analyzed fragments are interpreted as representations for the *still possible attribute values.* (For contrast, missing fragments in attribute grammars induce yet *undefined* attributes, which may propagate through evaluation of attribute equations.) The context-sensitive part of the language definition is expressed by equations between attributes, but without the directional constraints of attribute grammars (Section 5.1.1). This leads the authors to consider *context relations* between sets of attributes, rather than functional dependencies, as the fundamental tool in reasoning about the consistency of the program fragment.

Context relations are defined [BS86, Section 4.1] analogously to relations in data base theory [EN94]: For an abstract syntax tree with nodes N and attribute values A, a context relation $\mathrm{CR}(A)$ is a (possibly infinite) set $\{t\colon N \to A\}$. Each tuple t represents a simultaneous (and consistent) assignment of attributes to the nodes of the AST. The connection between subtrees is expressed with the relational [EN94] *natural join* operation: If a placeholder X in fragment F is replaced by a fragment G, yielding a new fragment H, we can compute $\mathrm{CR}(H) = \mathrm{CR}(G) \bowtie \mathrm{CR}(F)$. The natural join contains pairs of rows from the two relations that agree on the shared attributes. Context relations are conceptually infinite objects, they include all *still possible* attribute assignments. A finite representation for context relations is obtained by allowing variables in attributes and introducing unification to the join operation [BS86, Section 4.3]. The static semantics of language constructs and primitive operations is expressed by *basic relations* between attributes at the AST nodes of the constructs.

Context relations do not seem a natural tool for type analyses, and the authors planned to integrate a different formalism [Des84] in future versions [BS86, Section 2]. With types as attributes, the link between the declaration and reference of an identifier must be established prior to invoking context relations. Hence, scope analysis must be addressed separately, which Bahlke and Snelting mention as a short-coming [BS86, Section 4.3, Section 4.6]. Having settled this issue by external means, context relations naturally express overloading of predefined identifiers: Overloaded identifiers have basic relations with more than one tuple. However, this approach does not extend to user-defined functions, because each declaration has a single type. The solution is to generate basic relations from user declarations, which is reported to work in an experimental implementation [BS86, Section 4.3]. However, this procedure is again outside of the formalism.

5.1.3 ASF+SDF

Brand et al. [BHKO02] survey their ASF+SDF framework for language specifications. The Syntax Definition Formalism (SDF) describes the context-free and abstract syntax in a BNF-like formalism. The resulting parser is used to read the definition of the semantics in the Algebraic Specification Formalism (ASF). ASF is based on conditional rewriting rules, which are compiled directly to C-functions, using the `ATerm` library [BHKO02, Section 7.1] for efficient term data structures. The authors cite extensive experience with ASF+SDF in [BHKO02, Table I], including a number of type checkers. Among these, the checkers for CLaX [TD01] and Pascal [Deu96] are closest to our own work and appear representative of the framework's capabilities in this direction. Since CLaX is a Pascal dialect, we will restrict the discussion to the latter article.

Van Deursen [Deu96] contains the source of a static checker for a fragment of Pascal in ASF+SDF. We focus on Sections 4 and 5, which deal with environments and the type checks themselves. As an overall observation, the given specification has the flavor of functional programming, as the precise path of the computation is explicit in the rewrite rules. The following rule, for instance, type checks a constant declaration [Deu96, sec. 5.1.2].[1]

$$\frac{\begin{array}{r} \_id \text{ should not be declared in inner block of } E_2 = E_3 \\ \text{const-tc}(\_const, E_3) = E_4 \end{array}}{\text{const-defs-tc}(\_id = \_const, E_2) = E_4 + \text{const}\_id = \_const}$$

The phrase “should be” [Deu96, Section 4.4] refers to the expectation that the constant is undefined. The function returns the original environment if the expectation is met and extends the environment with an error message otherwise. As a second example, procedure calls are checked by the following rule [Deu96, Section 5.5.1].

$$\frac{\begin{array}{r} \text{find } \_id \text{ in } E_1.\text{context} = \mathit{Definition} \\ \mathit{Definition} \text{ defines } \_id \text{ as a normal procedure?} = \text{TRUE} \\ \text{call-tc}(\mathit{Definition}.\text{formal-parameters}, \_\mathit{act\text{-}par\text{-}list}, E_1) = E_2 \end{array}}{\text{proc\_call-tc}(\_id\ \_\mathit{act\text{-}par\text{-}list}, E_1) = E_2}$$

The predicates in the first two premises have to be defined literally by a recursive lookup in the environment E_1 (Sections 3.3 and 3.2.4).

Hendriks [Hen89a] shows how type inference for MiniML can be implemented in the ASF+SDF framework. As unification and substitution is not available at the ASF level, he must implement it by rewrite rules (Section 7.4.3). The type checker in Section 7.4.4 therefore resembles a purely functional checker (e.g. [Jon99]).

Dinesh and Tip [TD01] describe an elegant application of the framework to the explanation of type errors. Their rules for type checking rewrite a program fragment either to its type or to an error message. In the latter case, the rewrite relation can be examined backwards to find those program parts that have contributed to the subterm of the error message. The *dependency relation* between subterms in a single rewrite step can be extended to *dependency tracking* between the start and result terms [TD01, Section 4.2]. It yields the *slice*, a collection of subterms, of the program fragment that contributed to the error message.

5.1.4 CENTAUR and TYPOL

The CENTAUR system [BCD$^+$89] is a formalism for specifying the syntax and semantics of programming languages. It is based on natural semantics [Kah87] and includes a parser generator (*METAL*), a pretty printer (*Ppml*), an abstract data type for processing many-sorted ASTs (*VTP*), and the logical engine *TYPOL* [Des84]. TYPOL has been used to specify type inference for MiniML [CDDK86] and type checking for various Pascal-like languages [Des84, Section 6].

[1]The SDF translates the phrase “should not be declared in inner block of” to a term constructor.

The representation of ASTs is handled by the *virtual tree processor* (*VTP*) [BCD+89, Section 2]. It allows manipulation and persistent storage through an abstract interface and includes editing operations for interactive environments. Trees are many-sorted by the notion of *formalisms*. Each operator belongs to a *phylum* and must be declared with its operand phyla. Formalisms then are collections of phyla. A tree belongs to the formalism of its top operator, it can be constructed only if the subtrees' phyla match the operands' declared phyla. VTP can handle mutable positions in trees as *contexts* and *annotations* can be attached to tree nodes (such that attribute grammars can be handled by the VTP).

Both the static and dynamic semantics of a language are specified using the TYPOL component [Kah87, Des84]. The basic unit of specification is an *inference rule* with a *numerator* and *denominator*. Variables may occur in both and can be instantiated for each application of the rule. The numerator contains an unordered collection of *premises*. The formulae in the premises fall into two categories, *sequents* and *conditions*. Conditions are boolean expressions, which can be defined either again in TYPOL or by external procedures. A sequent has the form $A \vdash C$ where A is its *antecedent* and C its *consequent*; C must be a predicate, while A can have more general structure. (For example, one can store type assumptions, but this structure is not supported specifically by the formalism.) To structure the specifications further, rules can be grouped in named *rule sets*.

TYPOL specifications are compiled to Prolog for execution [Des84, Section 3.6]. Before the translation, the occurring syntax trees are type checked according to the formalisms of operators and types of variables are deduced. Furthermore, the arguments of predicates are classified as *in* or *out* (termed the arguments' *kinds*). The types and kinds of a predicate's arguments are used for overload resolution to avoid possible ambiguities in the generated Prolog program. The order of rules is determined by placing the best matching choice first, such that the more specific rules supersede the more general rules.

Type checking and -inference can be specified concisely in TYPOL [Des84, Sections 4, 5][Kah87, Section 6]. However, the specification resembles a logic programming formulation of the type checking algorithm, both in the lookup of variables in the environment and in the polymorphic `let` construct. The respective rules are [Kah87, Figure 8]:

$$\frac{\rho \overset{\text{type\_of}}{\vdash} \text{ident } x : \sigma}{\rho \vdash \text{ident } x : \tau} \quad (\tau = \mathit{inst}(\sigma))$$

$$\frac{\vdash P, \tau_2 : \rho' \quad \rho \vdash E_2 : \tau_2 \quad \rho + \rho'' \vdash E_1 : \tau_1}{\rho \vdash \text{let } P = E_2 \text{ in } E_1 : \tau_1} \quad (\rho'' = \mathit{gen}(\rho, \rho'))$$

The functions *gen* (for *generalization*), *inst* (for *instantiation*) and + (merging of environments with static scoping) must be programmed explicitly without support from the formalism itself. The *gen* predicate relies on native auxiliary predicates to extract the Prolog variables from Prolog terms, but the exact relation between type variables and Prolog variables is left implicit.

The rule set **TYPE_OF** describes a linear search through the environment. Scoping and shadowing of variables are implicit in the order of the entries.

$$\rho \cdot x : \sigma \vdash x : \sigma \qquad \frac{\rho \vdash x : \sigma}{\rho \cdot y : \sigma' \vdash x : \sigma}$$

The exact relation between natural semantics as a formalism and the executable Prolog program are left unspecific. Indeed, the intuitive isomorphism fails as Prolog interpretations omit the occurs check in unification: According to the remarks in [Kah87, Section 6.4], the type inferencer for MiniML erroneously assigns a recursive type to the expression $\lambda x.x\,x$, which is not typable in ML's basic type discipline due to the constraint $\tau = \tau \to \tau'$ for any type-assignment $x : \tau$ and some τ'.

5.1.5 Tinker Type

TinkerType [LP03] is a language to manage independent parts of formal systems, which are not necessarily type checkers, and assemble them into a complete system by selection. Its formalism is based on *features* and *clauses* [LP03, Section 2]:

$$\begin{aligned}&\textbf{Fts} \text{ a set of features}\\&\textbf{Cls} \subseteq \textbf{Names} \times \mathcal{P}(\textbf{Fts}) \times \textbf{Cnt}\end{aligned}$$

The features **Fts** and **Names** a symbols (the name of a clause is also called a *label*). The content **Cnt** of a clause is an uninterpreted string. Dependencies between features are specified by a *dependency* relation

$$\textbf{Dep} \subseteq \mathcal{P}(\textbf{Fts}) \times \mathcal{P}(\textbf{Fts})$$

For a given set of features F, the $closure(F)$ is the least superset F' of F that is closed under the dependency relation. A set F *dominates* a set F' if $closure(F) \supseteq closure(F')$.

The assembly of a formal system for selected features F_0 is guided by the feature annotations of the clauses: A clause $\langle n, F, c\rangle$ is eligible for inclusion if $closure(F_0) \supseteq closure(F)$, that is the requested features include the features for which the clause is relevant (under the dependency relation). If several clauses with the same name are eligible for inclusion, the clause with the maximal set of features (with respect to $\supseteq$) is chosen, if it exists. If the maximum does not exist, an error is signaled. This procedure ensures that the most specific applicable clause is chosen (cf. [LP03, Section 3.1]).

Features also implement a consistency check based on a set of propositional formulae over features that must be true for the set F_0 of selected features. For instance, to capture that in some variants (*kfsub*/*ffsub*) of $F_{<:}$ (see also [Car93]) joins cannot be computed (feature *calcjoin*), but have to be supplied by the programmer (feature *anotjoin*), the following formulae are checked during assembly [LP03, Section 5.6]:

$$\begin{gathered}sub \wedge (bool \vee variant) \Rightarrow calcjoin \oplus anotjoin\\ \neg(kfsub \wedge ffsub)\\ \neg(ffsub \wedge calcjoin)\end{gathered}$$

One of the applications of TinkerType is the assembly of a type checker for a given set of features of a type system. The clauses then contain branches of a Caml `match` statement like the following [LP03, Section 4] (where `T-If` is the clause's name and `{#` and `#}` delimit its verbatim content).

```
T-If
{#TmIf(fi,s1,s2,s3) ->
  if tyeqv ctx (typeof ctx s) TyBool then
     let tyS = typeof ctx s2 in
        if tyeqv ctx tyS (typeof ctx s3)  then tyS
        else error fi "arms of conditional have different type"
 else error fi "guard of conditional not a boolean"#}
```

This fragment computes the types of the guard expression `s1` and the two arms `s2`, `s3` and checks that `s1` has type `bool` and `s2`, `s3` have the same type (all checks are modulo convertibility as is necessary in higher-order type systems [Bar91]). The result is the type of `s2`.

The approach of TinkerType to the generation of type checkers is to represent a variety of checkers that are alike in their conventional implementation and that can be composed by concatenating single clauses textually. The feature dependencies, in their role of consistency checks [LP03, Figure 3–6, Section 5], determine which clauses cannot or must be applied together. A similar mechanism may be useful if Tcg libraries (Section 4.2) are built and deployed by different users.

5.2 Constraints

A well-studied abstraction views type systems as the connection of two largely independent components: A recursive traversal of the program's parse tree generates a set of *constraints* and the program is well-typed if these constraints have a solution — the type checking and type inference problems are reduced to the constraint solution problem.

In this light, Hindley-Milner type inference is reduced to equality constraints over the Herbrand universe of types [Wan87] and type inference with subtyping is reduced to set containment [Mit91, AW93]. By encoding objects as records of functions [Red88], subtyping also yields type inference for object-oriented languages (and recursive types) [EST95a, EST95b]. By considering type classes [WB89] as constraints, overloading can be treated in a uniform manner [Jon94, Jon95]. A restricted form of dependent types allows integer constraints over the types' parameters [XP99]. More generally speaking, constraints allow the designer of a type system to factor the essential consistency conditions that the type system imposes on the language from the treatment of standard language constructs.

Solving constraints in this context does not necessarily mean exhibiting a syntactic substitution of types for variables such that constraints are satisfied. Instead, it may also mean checking for *satisfiability*, which is given by the existence of a *valuation*[2] (in the sense of logic, e.g. [Gal86, Sections 3.3, 5.3]) assigning (semantic) types to type variables and interpreting the constructors of type expressions as functions on types. In the case of subtyping, for example, Aiken and Wimmers [AW93, Section 2] distinguish syntactic type expressions from types, which are ideals in a semantic domain [MPS86]. The solution of a constraint problem is then a valuation assigning types to variables, such that all subtyping

[2]This term is not standard in the literature, the term "substitution" is often used. Because substitution is a strictly syntactic notion in the remainder of this thesis, I prefer to introduce "valuation" instead.

constraints are satisfied. Instead of exhibiting solutions, the type inferencer only checks for their existence. This is achieved [AW93, Section 7] by successively transforming a constraint system to a simpler form such that the set of solutions does not change, but such that the result is satisfiable by its structure. Simplification for subtyping constraints is done (essentially) by *decomposition*[3] and *transitivity* (similar to first-order unification [MM82, BS01]). Decomposition simplifies for example a constraint $s \to t \le s' \to t'$ into two constraints $s' \le s$ (contravariance) and $t \le t'$. Transitivity adds for two constraints $s \le \alpha, \alpha \le t$ a new constraint $s \le t$. Satisfiability is then checked indirectly. Aiken and Wimmers prove that if a constraint set is fully simplified, that is no simplification rule applies, and it does not contain a constraint that is unsatisfiable (or *inconsistent*) by incompatible type constructors, then the constraint set is satisfiable.

Since constraint sets in subtyping grow linearly with the program size, a proper notion of *simplification* is required that effectively reduces, rather than enlarges, the constraint set. Pottier [Pot98, Pot01] gives such a method. (Eifrig et al. [EST95b, Appendix A] propose a precursor.) Again, his simplification is based on *closure of constraints* under decomposition and transitivity to check for satisfiability. Using the notions of *reachability* and *polarity*, he removes constraints that do not restrict the result type of the expression under consideration — these constraints can be discarded without influencing the solutions of the constraint system in an essential manner.

This exemplary introduction already clarifies the relation of constraint-based type systems to Tcg: Briefly put, Tcg is a framework to formalize specifically that part of typing problems that constraint-based systems assume as given, namely the recursive traversal of programs. Rather than taking the involved typing rules as straightforward, it investigates their structure. Tcg cannot be, and therefore does not aim to be, a general mechanism for *solving* constraints. (Although for example the decomposition rules of subtype closure can be captured, the complementary transitivity rule would require multi-headed[4] Tcg rules, which again would contradict the tree structure of proofs.

To give a more detailed impression of this claimed difference, I review briefly the HM(X) framework, a widely recognized abstraction from constraint-based type systems in Section 5.2.1 and survey several formulations of constraint solutions in Section 5.2.2.

5.2.1 The HM(X) Framework

Sulzmann [Sul00, OSW99] presents a framework HM(X) for type inference with constraints. The form of the occurring constraints is the parameter X, a *constraint system.* Sulzmann aims at identifying sufficient conditions on X for type inference with constraints from X to be sound and complete [Sul00, Sections 3.3, 4.3] for a λ-calculus with polymorphic *let* [Mil78].

A *cylindric constraint system* [Sul00, Section 3.1] is a structure

$$(\Omega, \vdash^e, \mathrm{Var}, \{\exists\alpha \mid \alpha \in \mathrm{Var}\})$$

in which Ω is the set of *tokens* or (primitive) constraints and $\vdash^e$ is entailment between (finite) sets of constraints. (These sets are also written as a logical conjunction $\wedge$ of

[3]The term is not used in [AW93], but in related literature.
[4]In the sense of [Frü98], see also Section 5.2.2.3.

primitive constraints.) Entailment satisfies $C \vdash^e D$ for each $D \in C$ and transitivity. The *projection operators*

$$\exists\alpha : \mathcal{P}(\Omega) \to \mathcal{P}(\Omega)$$

(for each variable α) must be compatible with entailment [Sul00, Definition 3] in the following sense.

$$C \vdash^e \exists\alpha.C \qquad C \vdash^e D \implies \exists\alpha.C \vdash^e \exists\alpha.D \qquad \exists\alpha.\{C, \exists\alpha.D\} =^e \{(\exists\alpha.C), (\exists\alpha.D)\}$$

A constraint system is *sound* [Sul00, Definition 9] if its projection operators can be interpreted as the existence of monotypes μ that satisfy the constraint.

The framework extends the Hindley-Milner calculus [Mil78, DM82], which allows polymorphism by type schemes $\sigma = \forall\alpha.\tau$ in judgments $\Gamma \vdash e : \sigma$. Sulzmann's generalization adds a constraint C to both the type schemes and judgments

$$\sigma = \forall\alpha.C \Rightarrow \tau \qquad C, \Gamma \vdash e : \sigma \ .$$

This formulation generalizes earlier work by Jones on qualified types [Jon95, Jon94], which again generalizes Haskell's type class formalism [WB89].

The main point of constraints in type derivations [Sul00, Section 3.2] is seen in the following typing rule. A part D of the "current" constraint C from a judgement can be transferred to a type scheme, thus restricting the possible instantiations of the bound variables $\tilde{\alpha}$. Furthermore, Sulzmann requires that there is at least one possible instantiation by keeping $\exists\tilde{\alpha}.D$, which inhibits unsatisfiable constraints in type schemes (cf. [Sul00, Section 3.5] for a comparison with other policies).

$$(\forall\ \text{Intro}) \quad \frac{C \wedge D, \Gamma \vdash e : \tau \quad \tilde{\alpha} \notin \mathbf{FV}(C) \cup \mathbf{FV}(\Gamma)}{C \wedge \exists\tilde{\alpha}.D, \Gamma \vdash e : \forall\tilde{\alpha}.D \Rightarrow \tau}$$

In the corresponding elimination rule, the constraint associated with the type scheme must be checked for the desired instantiation of the quantified variables.

$$(\forall\ \text{Elim}) \quad \frac{C, \Gamma \vdash e : \forall\tilde{\alpha}.D \Rightarrow \tau' \quad C \vdash^e [\tilde{\tau}/\tilde{\alpha}]D}{C, \Gamma \vdash e : [\tilde{\tau}/\tilde{\alpha}]\tau'}$$

Sulzmann's interpretation of type schemes justifies this reasoning [Sul00, Section 3.3.1]. The denotation of a type schema is the set of monotypes that are instances of the type τ and satisfy the constraint C.

$$\mathit{sem}(\forall\tilde{\alpha}.C \Rightarrow \tau) = \bigcap \{\mathit{sem}([\tilde{\mu}/\tilde{\alpha}]\tau) \mid \vdash [\tilde{\mu}/\tilde{\alpha}]C\}$$

The remainder of the typing rules merely propagate constraints. Proofs have a normal form [CDDK86, DM82] where ($\forall$ Intro) appears only at let-bound variables and ($\forall$ Elim) appears only at variable references [Sul00, Figure 3.2].

In Chapter 4, Sulzmann investigates type inference for the HM(X) framework. Whereas the logical, normalized typing rules (see [Sul00, Figure 3.2]) merely collect and eliminate constraints from different branches of the derivation, type inference also checks them for satisfiability and simplifies them. The central notion of this process is *normalization* [Sul00, Section 4.2], which abstracts from the *simplification* in the introduction to this Section 5.2. Sulzmann starts from a *term constraint system* with unitary unification, that is a system where (primitive) constraints are terms that have unitary unifiers. He first introduces a semi-lattice structure with ($\leq$, $\sqcup$) on *constraint problems*, which are pairs (C, ϕ) of constraints and substitutions. ($\phi_1 \sqcup \phi_2$ exploits the lattice structure on substitutions, see Section 2.1.2):

$$(D_1, \phi_1) \sqcup (D_2, \phi_2) = (\phi(D_1 \wedge D_2), \phi) \quad \text{where } \phi = \phi_1 \sqcup \phi_2$$

Given a constraint problem (D, ϕ), the least solved constraint problem (C, ψ) such that $(D, \phi) \leq (C\psi)$ is the *principal normal form* of (D, ϕ). Principal normal forms are unique up to $=^e$ (equivalence under constraint entailment) and *normalize* denotes the function from a constraint problem to its principal normal form (if it exists). Principal normal forms defined in this way are not necessarily equivalent up to variable renaming [Sul00, Definition 24, Lemma 24]. Type inference [Sul00, Figure 4.1] is then defined by collecting and normalizing the occurring constraints in each derivation step.

5.2.2 Solving Constraints

Although TCG does not aim to be a constraint solver, several of the analysis engines presented in this section are used for type inference and their merits relative to TCG must be clarified. The constraint handling rules in Section 5.2.2.3 in particular have generated some interest in describing type checkers in a declarative language. A second motivation for presenting a selection of available systems is a possible future integration to TCG (Section 6.2.3).

5.2.2.1 The Berkeley Analysis Engine (BANE)

Aiken et al. [AFFS98] present a toolkit BANE (Berkeley Analysis Engine) for constructing program analyses. BANE is a library that supports the solution of constraints over *sorts* `Term`, `FlowTerm`, and `Set`. Each sort s provides a constraint symbol $\subseteq_s$ and a solver for these constraints. The sort `Term` interprets $\subseteq$ as symmetric equality (optionally with recursive constraints). `FlowTerm` supports directed flow analyses with constructed values, `Set` enhances this capability with set intersection, union and a restricted complement [AW93]. All sorts have common simplification rules for transitivity and contravariant positions. Constraints over different sorts can be mixed by declaring constructor symbols with arguments from different sorts. In decomposing constraints, the corresponding constraint symbols are chosen for each argument position.

Constraints for a given program must be generated by a hand-written traversal function [AFFS98, Section 3] and by calls to the BANE constructor functions. After constraint solution, the result must be interpreted as the desired solution.

BANE is reported to perform equivalently to hand-written constraint solvers and to sometimes even outperform these due to specialized optimizations in the BANE solver.

5.2.2.2 Wallace and Dalton

Type inference with subtyping must solve the satisfiability problem. Although the problem does not permit an efficient algorithm in general (see [Sim03, Section 1.1] for a survey on the complexity of different subtyping problems), it can be argued that the constraints arising from practical programs may be handled with sufficiently optimized constraint solvers. Two such solvers, Wallace [Pot00] and Dalton [Sim03], are available as Caml libraries. Both libraries draw their efficiency from constraint simplification [Pot98, Pot01] which reduces the size of the constraint systems interleaved with solving the satisfiability problem.

5.2.2.3 Constraint handling rules

Constraint handling rules (CHR) are a formalism for specifying constraint solvers [Frü98]. Rules are divided into [Frü98, Definition 4.1] *simplification* rules

$$H_1 \,..\, H_i \texttt{<=>} G_1 \,..\, G_j \mid B_1 \,..\, B_k$$

and *propagation* rules[5]

$$H_1 \,..\, H_i \texttt{==>} G_1 \,..\, G_j \mid B_1 \,..\, B_k$$

In these rules, $H_1 .. H_n$ is the (multi-) *head*, $B_1 .. B_k$ is the *body* and $G_1 .. G_j$ are the *guards*.

Rules operate on a state $\langle F, E, D\rangle$, where F, E, and D are sets of constraints. F contains the *goals* yet be solved, E the constraints to be simplified by rules, and D the solved built-in constraints, including equality. The process continues until either F becomes empty (written as `true`) and E does not permit further simplification, or until D becomes `fail`. In the first case, the goals initially in F have been solved successfully, otherwise the constraints do not have a solution. Processing happens in four ways [Frü98, Definition 4.4]. *Solve* chooses a built-in constraint C from F and moves it to D, possibly simplifying $D \wedge C$. *Introduce* moves a constraint from F to E for further processing. *Simplify* chooses a simplification rule $H \texttt{<=>} G \mid B$ such that the head atoms H are unifiable with atoms $H' \subseteq E$ and G is satisfied from D and then introduces B to the F component of the result state. Thus the body of the rule contains new goals to be solved, while matched head atoms are removed from E. *Propagate* chooses propagation rule $H \texttt{==>} G \mid B$ and proceeds in the same way as *simplify*, except that it leaves the matched atoms H' in E instead of removing them.

Multi-headed rules with guards are essential to solving constraints [Frü98, Section 3.2]. With them, for example, one can specify the closure under transitivity as a propagation rule [Frü98, Section 2].

```
transitivity @ X=<Y, Y=<Z ==> X=<Z
```

Constraint handling rules have recently attracted interest for specifying and implementing constraint-based type inference [Sul01, SS01, SS04, AF02, AF04]. Sulzmann and Stuckey [Sul01, SS01, SS04] instantiate the HM(X) framework (see Section 5.2.1). In

[5]The *simpagation* rules $H_1 \,..\, H_l \setminus H_{l+1} \,..\, H_i \texttt{<=>} G_1 \,..\, G_j \mid B_1 \,..\, B_k$ in [Frü98, Definition 4.1] can be understood as an abbreviation [Frü98, Section 4.1].

HM(*CHR*) the constraint system can be given by user-defined constraint handling rules, while the type inference algorithm, including polymorphic let, remains fixed. The authors apply their framework to checking of metric units in numerical computations, overloading, record types and security protocols.

Alves and Florido [AF02, AF04] follow a similar path, but specify their type inference engine as Horn clauses to be executed in Prolog, for which CHR solvers are available [Frü98, Section 7]. Unfortunately their treatment of *let* polymorphism, a possible connection point with TCG's intentions, is not presented fully. Like the approach in Section 5.1.4, they represent type variables as Prolog variables and employ predicates `gen` and `inst` to handle type schemes, but they do not make their implementation precise.

5.3 Logical Frameworks

Logical frameworks [HHP93, Pfe96, Pfe01] offer environments for expressing logics and provide generic tools for reasoning in these logics. They reduce the effort necessary to build theorem provers and proof assistants for specific logics. By using well-tested core implementations they increase the confidence in the conducted proofs. The presentation in Section 5.3.1 is based on the survey by Pfenning [Pfe01]. The presented material has been selected based on similarities with TCG. To highlight the similarities and clarify the differences, we formalize type inference for MINIML in Isabelle [Pau94] in Section 5.3.2, including a novel approach to polymorphic let.

5.3.1 Overview

If a logical framework is to support reasoning in a given logic, the *object logic*, then three notions must be represented in the framework's *meta logic*: The object logic's (abstract) syntax, its derivations and its proof search. It is crucial that the representation is *adequate*, meaning that exactly the notions of the object logic can be represented and exactly the valid object-logic proofs can be conducted. (Some authors prefer to call the representation *faithful* if the representation does not permit theorems in the meta logic that are not derivable in the object logic and *adequate* if that representation is complete.) An adequate representation allows then to reason in the meta logic, but to interpret the reasoning in the object logic. The following subsections can but give a brief sketch of these considerations.

5.3.1.1 Syntax

First-order terms provide a direct representation function $\ulcorner\cdot\urcorner$ of object terms, given an enumeration $x_1, x_2, \ldots$ of the variables and constants c_f^k for all k-ary function symbols f.

$$\begin{aligned}\ulcorner x_n \urcorner &= \mathtt{var}(n)\\ \ulcorner f^k(t_1 \mathinner{..} t_k)\urcorner &= \mathsf{c}_f^k(\ulcorner t_1\urcorner \mathinner{..} \ulcorner t_k\urcorner)\end{aligned}$$

The representation of quantifiers is

$$\ulcorner \forall x.A\urcorner = \mathtt{forall}(\ulcorner x\urcorner, \ulcorner A\urcorner)$$

which requires an explicit treatment of bound names.

Adequacy of $\ulcorner\cdot\urcorner$ requires that exactly the object level terms can be represented. This does not hold in general because general first-order terms may have, for example, non-variable terms in the first argument of `forall` and a non-numeric argument to `var`. Therefore, adequacy must be ensured by some external means.[6] (Pfenning [Pfe01] uses a Horn-clause theory.) Furthermore, since we expect $\ulcorner A\urcorner \equiv \ulcorner A'\urcorner$ for $A \equiv A'$, equivalence modulo α-conversion must be implemented, also substitution must respect the bound names and must be specified explicitly. Where object logics involve bound names frequently and these must have their usual properties, a first-order representation is cumbersome.

Higher-order abstract syntax (HOAS) [PE88] solves this problem. It is based on a (simply) typed lambda calculus. A suitable choice of types ensures that only adequate terms can be constructed, while the meta-level λ-abstraction treats bound names. As a general principle, object-level bound names are represented by meta-level bound names. For example, the $\forall$ quantifier is represented by a constant `forall`.

$$\ulcorner\forall x.A\urcorner = \texttt{forall}(\lambda x.\ulcorner A\urcorner)$$

A substitution $A[t/x]$ is represented by a β-redex

$$(\lambda x.A)t$$

which again treats the necessary renamings in the meta logic.

For proof search, HOAS requires unification of λ-terms modulo β- and α-conversion. The problem of *higher-order unification* [Hue75, Dow01] is in general undecidable (although β-equivalence in the simply typed λ-calculus is decidable due to strong normalization). However, most concrete unification problems occurring in proof search have been found to be special cases [Pfe01][Pau94, Section 1.3.1]. The restriction to *higher-order patterns* [Mil91] has a deterministic unification algorithm (e.g. [Nip93]). Where the restriction does not apply, constraints between unknowns (*flex-flex constraints*) remain in the solution and these can be instantiated by the user explicitly in the interactive setting of proof assistants.

5.3.1.2 Derivations

To encode derivations, the inference rules of the object logic must be encoded in the framework's meta logic. The one closest to TCG seems to be intuitionistic natural deduction with Hereditary Harrop Formulae [MNPS91, Mil91]. This meta logic encodes inference rules as higher-order axioms [Pau86] with the following form:

$$H ::= P \mid \top \mid H_1 \wedge H_2 \mid H_1 \supset H_2 \mid \forall x.H \tag{5.3.1}$$

The essential differences to Horn clause logic are *embedded implication* and *embedded quantification* in the last cases. These permit parametric and hypothetical reasoning (Section 1.2.2) to be treated. As an example application of embedded implication, consider the encoding of ($\Rightarrow$I).

$$\forall A.\forall B.\big((\texttt{nd}(A) \supset \texttt{nd}(B)) \supset \texttt{nd}(A \Rightarrow B)\big)$$

[6]For an example in type inference see [Gas01].

The predicate `nd` denotes “derivable in natural deduction”, $\Rightarrow$ is the constant representing object-level implication and $\supset$ is meta-level implication. Embedded quantification can be observed in the encoding of ($\forall$I), which uses an embedded quantifier $\forall y$ to capture the *eigenvariable* condition on y (Section 1.2.2).

$$\forall A\big((\forall y.\mathtt{nd}(A\,y)) \supset \mathtt{nd}(\mathtt{forall}(\lambda x.A\,x))\big)$$

The meta-level λ-abstraction is taken from higher-order abstract syntax. The application $A\,x$ can be read as “A which possibly contains x”.

Miller et al. [MNPS91] observe that Hereditary Harrop Formulae allow for a direct proof search similar to SLD resolution on Horn clauses. This observation leads to the design of λProlog [Mil91, Fel93, NM98], a logic programming language with clauses from H.

Comparison Towards a comparison with Tcg, observe that quantification and implication are similar to those of nested Tcg rules (Section 2.2). Embedded quantification in H also plays the role of context modification (Section 2.2) in Tcg. For proof search (Section 5.3.1.3), meta-level assumptions are represented in sequents and meta-level implication is interpreted by adding the premise to the sequent’s antecedent (Section 1.2.2.2). These connections are not surprising, given that Tcg’s formalism is explicitly based on an analysis of logical calculi (Section 1.2.2).

Encoding Proofs The derivability considered so far does not yield derivations as objects. For many applications (e.g. [Nec97]) it is desirable to have derivations encoded, for example for verifying the constructed proof after it is completed, thus increasing the confidence in its correctness. Dependently typed λ-calculi form a basis for this encoding (e.g. [HHP93]), using the Curry-Howard isomorphism [How80, CH88, ML84, Bar91]. Dependent types arise from meta-level quantification (written as Π).

$$\frac{\Gamma \vdash A : \mathtt{type} \qquad \Gamma \vdash \Gamma, x : A \vdash M : B}{\Gamma \vdash (\lambda x : A.M) : \Pi x : A.B}$$

5.3.1.3 Proof Search

Once a logic is captured in a framework, a method to construct proofs is required. Since the application is mostly an interactive environment, proof search does not need to be automatic nor complete — the user can guide the framework to find proofs. A widespread approach is to have the user apply a *tactic*, which is a function from theorems to theorems, where a theorem represents a partially completed proof. The tactic’s result theorem is constructed such that whenever it is proven, then the input theorem is also proven. (A tactic may also return alternative proof attempts [Pau94], any of which can be completed to complete the input proof.) In the original proposal [GMW79], *theorem* is an abstract data type and tactics are the only operations on that type. If the tactics allow only logically valid inference steps to be conducted, then any constructed proof will be valid as well. In the setting of Hereditary Harrop Formulae, tactics resolve open goals in a sequent-style meta-logic [Pfe01, Section 4.1][Pau94, Chapter 6].

Frequently, a sequence of tactics re-occurs in interactive proofs several times. It is then desirable to name that tactic and refer to the name later on. Following [GMW79], tactics are usually programmed in a meta-level programming language and that language's binding constructs can be used to compose and name tactics. Using higher-order programming [GMW79, Fel93] frequent patterns of composition can also be coded and named. These compositions are termed *tacticals*, they are functions from tactics to tactics. For example, t_1 `ORELSE` t_2 applies tactic t_1. If it succeeds, its result is the result of the entire tactic. Otherwise, the result is obtained by applying t_2. Similar constructs can be given for sequencing and repetition (e.g. [Pau94, Chapter 7]) and new tacticals can be provided in the meta-language if the need arises.

5.3.2 MiniML in Isabelle – A Case Study

This section contains the development of type inference for MiniML [CDDK86] in Isabelle [Pau89, Pau94]. The language includes polymorphic *let*, which is commonly believed to be fundamentally incompatible with higher-order abstract syntax [Pfe88, DP91, Han98, Lia97, Lia02]. However, a closer examination in Section 5.3.2.3 shows that the problematic *generalization* operation can be accomplished using the definition of higher-order pattern unification [Mil91, Nip93]. The Isabelle source code in this section is deliberately shown in plain textual form, disregarding the presentation facilities for theories [NPW02, Chapter 4]. I hope to make more clear in this manner the actual working of the formalization.

5.3.2.1 Syntax

Isabelle employs a polymorphically typed λ-calculus for terms. We introduce new types for each category of terms we would like to distinguish: Types `ty`, expressions `exp`, type schemes `tys` are canonical, the auxiliary `index` and `tylst` allow a single type scheme to quantify over a set of type variables in Section 5.3.2.3. Finally, the object-level propositions `o` represent typing judgments.

```
types
  o
  ty
  exp
  tys
  index
  tylst
```

The terms of these types are specified in Isabelle by *constants* denoting term constructors. We start to define the types `ty` with the following constants. (Further ones can obviously be added if desired. The `infixr` keyword is described below.)

```
  int  :: ty
  bool :: ty
  fun  :: "[ ty, ty ] => ty" (infixr "->" 40)
```

The indexes to type lists are represented by constants `zi` (*zero index*) and `ni` (*next index*).

```
zi  :: index
ni  :: "index => index"
```

The type list constants[7] have *mixfix declarations*. The numerical annotations are *priorities* [Pau94, Section 10.1] which establish precedences of non-terminals. If non-terminal A derives sentence γ with priority p, then that derivation can be used in some production $B \rightarrow \cdots A^{(q)} \cdots$, where A has priority q, if $p \geq q$. A left-associative infix operator $\circ$ with precedence p can thus be realized by a production $A^{(p)} \rightarrow A^{(p)} \circ A^{(p+1)}$ and likewise for right-associative operators. The keywords `infixl` and `infixr` abbreviate these productions.

```
cons :: "[ ty, tylst ] => tylst" (infixr "++"  70)
nil  :: "tylst"                  ("[]" 1000)
```

On type lists we also provide an auxiliary constant `nth` denoting access to some element in the list. Predicate `get_nth` will later be axiomatized to actually yield that type from a specific list constant.

```
nth     :: "[ tylst, index ] => ty"
get_nth :: "[ tylst, index, ty ] => o"
```

MiniML expressions are also constants with mixfix syntax as specified by the following declarations.

```
If   :: "[ exp, exp, exp ] => exp" ("if _ then _ else _" [ 0,41,40 ] 40)
Lam  :: "(exp => exp) => exp" (binder "lam" 11)
App  :: "[ exp, exp ] => exp" (infixl "‘" 30)
LET  :: "[ exp, tys, (exp => exp) ] => exp"
```

Constants `If` and `App` are straightforward. (We have to add an auxiliary operator ‘ for curried application to avoid ambiguities with meta-level function application in Isabelle's `Pure` theory.) The constant `Lam` represents an abstraction $\lambda x.e$ in higher-order abstract syntax by a term `Lam(%`x`.`e`)` (where `%` is Isabelle's meta-level λ-abstraction, see also Section 5.3.2.1, [Pfe01, Section 2.2]). The declaration `binder` in this constant's declaration tells Isabelle to generate the necessary translations from external to internal presentation and vice versa. In case of constant `LET` with its special sequence of arguments, we must provide translations explicitly. `LET` takes as arguments the bound expression, its inferred type scheme and the body expression. The bound variable is implicit in the λ-abstraction on the *let* body. To provide the usual external syntax for the programmer, we first introduce an auxilary constant `@LET`, which will only be created temporarily during the parse process and whose external presentation is the usual one. Note that the bound variable is an explicit parameter of that constant, that is the temporary representation is first order.

[7]Theory `MiniML` is derived from `Pure`, the theory containing only Isabelle's meta-logic, such that polymorphic lists as in Isabelle/HOL are not available. To keep the presentation simple, we introduce only the required monomorphic instance.

```
syntax
  "@LET" :: "[ exp, ty, exp, exp ] => exp"
            ("let _ : _  = _ in _" [ 0,0,0,0 ] 29)
```

The intermediate presentation must then be translated to the internal form `LET`. The λ-abstraction contained in the rewrite rule effects the name capture on the bound variable.

```
translations
  (exp) "let x : t = e1 in e2" == (exp) "LET(e1, t, %x. e2)"
```

A similar procedure, albeit without bound names, applies to the infix notation for `+` and `*`. The intermediate constants take two parameters.

```
  "@plus" :: "[ exp, exp ] => exp" (infixl "+" 15)
  "@mul"  :: "[ exp, exp ] => exp" (infixl "*" 16)
```

These constants have two parameters, but type checking works on expressions with curried application. Two further translations thus pre-process the parse tree to use the (also declared) constants `cplus` and `cmul`.

```
  (exp) "e1 + e2" == (exp) "cplus ‘ e1 ‘ e2"
  (exp) "e1 * e2" == (exp) "cmul  ‘ e1 ‘ e2"
```

Integer constants are embedded to the expressions by the constant `ilit` (for *integer literal*). The mixfix annotation `"_"` allows Isabelle to convert the built-in numbers to expressions during parsing.

```
  ilit :: num  => exp           ("_" 1000)
```

There are two type assignment operators. The first one is used for typing judgements and λ-bound identifiers. The second one assigns a type scheme, rather than a type, to an identifier. We use a different symbol to avoid ambiguities.

```
  typeof :: "[ exp, ty  ] => o" (infixl ":" 10)
  tysof  :: "[ exp, tys ] => o" (infixl ":*" 10)
```

Finally, the object level propositions are identified with meta-level propositions by the constant `Trueprop`, which represents derivability in the sense of [Pfe01, Section 3] (see also Section 5.3.1.2, Remark 1.2.1). We use the mixfix notation `"_"` again to insert `Trueprop` automatically where necessary.

```
  Trueprop :: "o => prop" ("(_)" 7)
```

5.3.2.2 Simply Typed λ-Calculus

Type checking rules are represented by ISABELLE theorems, which are named Hereditary Harrop Formulae (Section 5.3.1). For example, function application

$$\frac{\Gamma \vdash f : s \to t \quad \Gamma \vdash e : s}{\Gamma \vdash f\, e : t}$$

becomes the theorem `funE`.[8]

```
funE "[| f : s -> t; e : s |] ==> f ‘ e : t"
```

In a similar manner, the constant `If` is checked by

```
tyif  "[| a : bool; b : t; c : t |] ==> if a then b else c : t"
```

The typing rule for λ-abstraction

$$\frac{\Gamma, x : s \vdash e : t}{\Gamma \vdash \lambda x.e : s \to t}$$

involves higher-order abstract syntax and inner quantification.

```
funI "(!! y . ( y : s ==> (e(y) : t)))
      ==> lam x . e(x) : s -> t"
```

We can read the consequence `lam x. e(x)` directly as "a λ-abstraction, where `x` may occur in `e`". When the rule is applied, the conclusion has been parsed into the internal form `Lam(%x.`e`(x))`. To solve a goal `Lam(%z.E)`, where `z` may occur in E, higher-order unification [Hue75, Nip93, Dow01] recursively matches the term structure of E against variable e. The meta-level treatment of bound names ensures that the bound occurrences of `z` in E become bound occurrences of `x` in e.

After this unification is complete, `%x.`e(`x`) is instantiated to a λ-abstraction α-equivalent to `%z.`E. The premise of `funI` exploits this representation. It introduces a fresh name `y` (an *eigenvariable* in the sense of Section 1.2.2) and establishes the new goal

```
y : s ==> e(y) : t
```

The application `e(y)` after β-reduction reveals the term structure of the body expression `E` again, but all references to the bound `z` are replaced by the fresh `y`.

The other constants can be trivially checked, we only mention them for completeness.

```
tyilit "ilit (i :: num)  : int"
typlus "cplus : int -> int -> int"
tymul  "cmul  : int -> int -> int"
```

In this setup, type inference for the simply typed λ calculus can be conducted by repeatedly applying the given typing rules to the first open goal until no goals remain. In this process, ISABELLE's unknowns (existentially quantified variables) play the role of type variables to be instantiated during the inference.

[8]`==>` denotes meta-level implication, the brackets `[|` and `|]` surround multiple premises, which are separated by `;`. The notation `:` is parsed to constant `typeof` defined above, `‘` is `App`. All identifiers that are not declared as constants are considered variables and these are quantified at the theorem. Equivalently, we could have prefixed the expression by `!!f,e,s,t`, representing meta-level quantification.

Instantiation Instantiation of type schemes occurs at variables references [DM82]. The corresponding ISABELLE rule is (binder ALL is introduced in Section 5.3.2.3):

```
var "[| x :* ALL a . t(a); inst(t(bs),t') |] ==> x : t'"
```

Note how in the type scheme `t` the bound name `a` is replaced by a fresh unknown `bs`, representing a list of type unknowns. The predicate `inst` produces a copy of `t`, where all references by `nth` have been replaced by fresh type unknowns. The case distinction is straightforward, except for `inst_var`. That rule tentatively unifies its argument with a constant `TYV`, which occurs nowhere else. Hence, the unification succeeds iff it meets an unknown (which represents a type variable).[9]

```
inst_nth "get_nth(bs,i,ty) ==> inst(nth(bs,i),ty)"
inst_var "inst(TYV(a),a)"
inst_int "inst(int, int)"
inst_bool "inst(bool, bool)"
inst_fun "[| inst(s,s'); inst(t,t') |] ==> inst(s -> t, s' -> t')"
```

We remove the auxiliary constructor `TYV` later on by rewriting with

```
remove_TYV "TYV(a) == a"
```

The auxiliary predicate `get_nth` uses a standard Prolog programming technique: It instantiates its second parameter, a list of fresh unknowns, *while* it retrieves a desired entry. Hence the first time that entry i is accessed, a new unknown is created. All later references to i return the same unknown.

```
get_nth_zi "get_nth(a ++ as, zi, a)"
get_nth_ni "get_nth(as,i,a) ==> get_nth(b ++ as, ni(i), a)"
```

5.3.2.3 Polymorphic *let*

Consider now the rule for polymorphic *let*

$$\frac{\Gamma \vdash e : s \quad \tilde{\alpha} = \mathbf{FV}(s) \setminus \mathbf{FV}(\Gamma) \quad \Gamma, x : \forall\tilde{\alpha}.s \vdash e' : t}{\Gamma \vdash \mathbf{let}\ x = e\ \mathbf{in}\ e' : t} \text{(let)}$$

We will represent type schemes in the canonical way [Pfe01] by mapping quantified type variables to meta-level bound names. (This is the same procedure as for λ bound names in `Lam`.) We represent the vector $\tilde{\alpha}$ by a single bound name `z` which is indexed to provide simultaneous abstraction over several variables.

```
All :: "(tylst => ty) => tys" (binder "ALL" 20)
```

A direct specification of (let) fails:

[9]An alternative solution consists in using `match_tac` [Pau94, Section 6.1.3] with the rules for type constants. Matching fails if any unknowns in the goal would be instantiated. If all constants are tried, only variables can remain.

```
letI0 "[|
          e : s;
          gen(z,zi,s,li, s');
          (!!y. (y :* All(s')) ==> e'(y) : t)
      |] ==> let x : All(s') = e in e'(x) : t"
```

In this rendering, the premise requests name capture of the $\tilde{\alpha}$ in the generalized type `s'` — an operation that is carefully avoided in higher-order abstract syntax. It has been observed [Han98, Lia97, Lia02] that this is a fundamental obstacle to programming ML type inference in languages with HOAS, in particular λPROLOG [NM98]. Furthermore, the second premise requires a computation with *meta-level* objects, the unknowns contained in s and Γ. I now present the solution to both problems.

Put the other way around, we must enable the type s in (let) to contain the bound variables $\tilde{\alpha}$. In ISABELLE terminology, this means that we must apply the variable `s` to the bound name `z`, in the same way that we have applied `e(x)` in the conclusion of `funI` to allow the bound `x` to appear in `e`. The first premise of the `letI` theorem exhibits this strategy: It surrounds both the type inference on `e` and the generalization with a meta-level quantifier that introduces a new name `z` standing for the vector $\tilde{\alpha}$. Now `z` may appear in `s` and `s'`, the generalized version of `s`.[10]

```
letI "[|
          (!! z .((e : s(z)) &
                   (gen(z,zi,s(z),li, s'(z)))));
          (!!y. (y :* All(s')) ==> e'(y) : t)
      |] ==> let x : All(s') = e in e'(x) : t"
```

The second premise and the conclusion follow the pattern of `funI` above, except for `All(s')` which notes down the inferred type scheme for later checks.

Identifying Generic Variables We will employ the treatment of scopes in higher-order pattern unification [Nip93] to identify the generic variables $\tilde{\alpha}$ in (let).[11] Unification treats scoping by applying unknowns to those bound names that may legitimately occur in substitutions for the unknown.[12] When u occurs nested in some λ-abstractions

$$\lambda y_1 \mathinner{..} \lambda y_n.(\cdots u(\tilde{z}) \cdots)$$

then $\tilde{z} \subseteq \{y_1 \mathinner{..} y_n\}$, because only the y_i may occur in substitutions for u. If u also occurs in a smaller context

$$\lambda y_1 \mathinner{..} \lambda y_{n-k}.(\cdots u(\tilde{z}) \cdots)$$

[10]The auxiliary symbol $P\&Q$ is not part of ISABELLE's meta-logic, but is introduced as an abbreviation for $P \Longrightarrow Q \Longrightarrow$ `true`.

[11]Rémy [Rém92, Section 2] proposes a similar identification scheme to improve the efficiency of ML type inference. He assigns an integer, the *rank*, to each type variable and typing judgment such that the variable can be generalized iff its rank is equal to the rank of the first `let` premise. It is instructive to compare his proofs to the ones given below.

[12]Miller [Mil92] proposes a different approach.

then $\tilde{z} \subseteq \{y_1 \mathinner{\ldotp\ldotp} y_{n-k}\} \subset \{y_1 \mathinner{\ldotp\ldotp} y_n\}$. By this reasoning, the bound names $\tilde{z}$ given indicate the highest nesting level at which u occurs.

Consider now again the meta-level quantification `!!z` in the first premise of `letI`. Goals in ISABELLE are represented by closed terms of the form [Pau94, Section 1.5.2]

$$\forall \tilde{x}. [\![\phi_1 \mathinner{\ldotp\ldotp} \phi_n]\!] \implies \phi$$

At each `!!z` quantification, the set $\tilde{x}$ is enlarged. Hence, like in the λ-term above, the bound names at every unknown indicate at which *outermost* nesting `let`-level the unknown occurs, and this is precisely the information required by the `gen` predicate.

In the remainder of this section, we will also use for brevity

$$(u \in t) := (u \in \mathbf{FV}(t)) \qquad \text{and} \qquad u(\tilde{z}) \in t$$

to say that u occurs in t and is applied to $\tilde{z}$ in this occurrence.[13]

The goal `gen` in `letI` has two auxiliary parameters `zi` and `li`, which implement the numbering of several generalized variables. Let us assume for the theoretical development that `gen` could instantiate s' directly with a λ abstraction.

$$s' \mapsto \%\tilde{u}.s \quad \text{where } \tilde{u} = \{u(\tilde{z}) \in s \mid \mathtt{z} \in \tilde{z}\} \tag{5.3.2}$$

We will see later how this instantiation is effected in ISABELLE.

Correctness of `gen` Let us confuse types and type schemes for the moment, because they play similar roles in the subsequent development. Then our typing judgments have the form

$$\forall \tilde{x}. [\![y_1 : s_1 \mathinner{\ldotp\ldotp} y_n : s_n]\!] \implies e : t$$

Consider now the nesting of `lam` expressions and expressions in the first premise of `let`. Each `let` application introduces a new name `z` into $\tilde{x}$. (We disregard the `y` added to $\tilde{x}$ by `funI`.) Let us mentally annotate each y_k with that subset $\tilde{x}_k \subseteq \tilde{x}$ that was noted at the typing judgment that added y_k to the type assumptions. We then have judgments

$$\forall \tilde{x}. [\![y_1^{\tilde{x}_1} : s_1 \mathinner{\ldotp\ldotp} y_n^{\tilde{x}_n} : s_n]\!] \implies e : t$$

5.3.1 DEFINITION. Let

$$J = \forall \tilde{x}. [\![y_1^{\tilde{x}_1} : t_1 \mathinner{\ldotp\ldotp} y_n^{\tilde{x}_n} : t_n]\!] \implies e : t_{n+1}$$

be a typing judgement and let $\tilde{x}_{n+1} := \tilde{x}$. We say that J is a *proper* typing judgment iff for all $i \in 1 \mathinner{\ldotp\ldotp} n+1$ and $u(\tilde{z}) \in t_i$

$$\tilde{z} \subseteq \tilde{x}_i \quad \text{and} \quad \tilde{z} \subset \tilde{x}_i \iff \exists y_j^{\tilde{x}_j}\ \tilde{x}_j \subset \tilde{x}_i \wedge u \in t_j \tag{5.3.3}$$

[13]A more common notation is $t[u(\tilde{z})]$, where t denotes a *context*, which is a term with a designated place containing $u(\tilde{z})$. We feel that the chosen $\in$ notation expresses better that u is a free variable in t in the first place and is applied to $\tilde{z}$ additionally.

5.3.2 PROPOSITION. *Let J be a proper unsolved typing judgment and $J\sigma$ be the judgment once it has been proven by the typing theorems. Then $J\sigma$ is a proper typing judgement.*

Proof. We use the notation from Definition 5.3.1 for the assumption that J is a proper typing judgement. We have to show that for all $i \in 1 \mathinner{..} n+1$ and $v(\tilde{z}') \in t_i\sigma$

$$\tilde{z}' \subseteq \tilde{x}_i \quad \text{and} \quad \tilde{z}' \subset \tilde{x}_i \iff \exists y_j^{\tilde{x}_j}\ \tilde{x}_j \subset \tilde{x}_i \wedge v \in t_j\sigma \tag{5.3.4}$$

Application of substitutions respects bound names, applying σ never enlarges the set of bound names at a variable [Hue75, Nip93]. Therefore, the first part of (5.3.4) is immediate by assumption (5.3.3). For the second part, fix some $v(\tilde{z}') \in t_i\sigma$ such that $v \in t_j\sigma$ with $\tilde{x}_j \subset \tilde{x}_i$. We show that $\tilde{z}' \subset \tilde{x}_i$. There must be some $u_i(\tilde{z}_i) \in t_i$ and $u_j(\tilde{z}_j) \in t_j$ with[14] $\sigma(u_i) = s_i[v]$ and $\sigma(u_j) = s_j[v]$ (possibly $u_i = v$ and/or $u_j = v$). We have $\tilde{z}' \subseteq \tilde{z}_j$ by definition of substitution, $\tilde{z}_j \subseteq x_j$ by (5.3.3) and finally $\tilde{x}_j \subset \tilde{x}_i$ by assumption.

The converse is proven by induction on the size of the proof. If the proof is by using a typing assumption, then t_{n+1} is unified with one of the $t_1 \mathinner{..} t_n$ and σ is the unifier. Fix some $v(\tilde{z}') \in t_i\sigma$ such that $\tilde{z}' \subset \tilde{x}_i$. We show that there is some y_j such that $\tilde{x}_j \subset \tilde{x}_i$ and $v \in t_j\sigma$. By definition of unification, there is a maximal set $\{u_k(\tilde{z}_k) \in t_{i_k} \mid u_k \neq v, \sigma(u_k) = s_k[v]\}_{k\in K}$. If the set is empty, then $v \in t_i$ and we are done by (5.3.3). Otherwise, by definition of unification, we have $\tilde{z}' = \bigcap_{k\in K} \tilde{z}_k$. Since $\tilde{z}' \subset \tilde{x}_i$, the set $\{k \mid z_k \subset \tilde{x}_i\}$ is not empty. Select that k such that $\tilde{x}_{i_k}$ is minimal. (The minimum exists because the y_i have been inserted linearly, hence the $\tilde{x}_i$ are linearly ordered w.r.t. $\subseteq$.) Then we have $\tilde{z}_k = \tilde{x}_{i_k}$. (Suppose this was not the case, that is $\tilde{z}_k \subset \tilde{x}_{i_k}$. Then by assumption (5.3.3), there was some $t_{i_{k'}}[u_k]$ with some yet smaller $\tilde{x}_{i_{k'}}$, contradicting minimality of $\tilde{x}_{i_k}$.) With this result, we have

$$\tilde{x}_{i_k} = \tilde{z}_k \overset{\text{choice of } \tilde{z}_k}{\subset} \tilde{x}_i$$

The same reasoning applies for `var` with `inst`, since `inst` only introduces fresh variables, which are annotated by $\tilde{x}$.

At rule `funI`, we prove a goal

$$J' = \forall\tilde{x}.[\![\phi_1 \mathinner{..} \phi_n]\!] \Longrightarrow \texttt{lam}\ y.e : t_{n+1} \to t_{n+2}$$

where (5.3.3) holds. The premise of `funI` generates a goal[15]

$$J = \forall\tilde{x}.[\![\phi_1 \mathinner{..} \phi_n, y^{\tilde{x}} : t_{n+1}]\!] \Longrightarrow e : t_{n+2}$$

We have only shifted t_{n+1} by one position, hence that goal again satisfies (5.3.3). By induction hypothesis, the instance $J\sigma$ satisfies (5.3.4). We show that $J'\sigma$ satisfies (5.3.4). The claim is clear for any $i \leq n$ by induction hypothesis. For $i = n+1$ and $i = n+2$ observe that these types are inserted under the same $\tilde{x}_{n+1} = \tilde{x}_{n+2} = \tilde{x}$ and use the induction hypothesis.

For `funE`, we solve

$$J' = \forall\tilde{x}.[\![\phi_1 \mathinner{..} \phi_n]\!] \Longrightarrow f\,\text{`}\,e : t$$

[14] Notation for contexts s_i and s_j.

[15] Recall that we include in $\tilde{x}$ only the `z` introduced in `let`.

where (5.3.3) holds. The goals J_1, J_2 we generate obviously satisfy (5.3.3), hence, the proof for $J'\sigma$ also contains two proofs for $J_1\sigma$, $J_2\sigma$, for which (5.3.4) holds by induction hypothesis. But then (5.3.4) is immediate for $J'\sigma$. Rule `tyif` and the rules for constants are treated likewise.

For `letI`, we prove (we omit the type scheme annotation for y for brevity)

$$J' = \forall\tilde{x}.[\![\phi_1 \mathinner{..} \phi_n]\!] \Longrightarrow \texttt{let}\ y = e\ \texttt{in}\ e' : t$$

where (5.3.3) holds. The two main judgments are

$$J_1 = \forall\tilde{x},\texttt{z}.[\![\phi_1 \mathinner{..} \phi_n]\!] \Longrightarrow e : s(\tilde{x},\texttt{z}) \qquad J_2 = \forall\tilde{x}.[\![\phi_1 \mathinner{..} \phi_n, y^{\tilde{x}} : \texttt{All}(s')]\!] \Longrightarrow e' : t$$

Suppose we are given a proof of $J'\sigma$. It contains proofs for $J_1\sigma$ and $J_2\sigma$. Both J_1 and J_2 satisfy (5.3.3) directly: For the $\phi_1 \mathinner{..} \phi_n$, we use (5.3.3), and the unknown s fulfills (5.3.3). For J_2, observe that the abstraction s', not the application $s'(\texttt{z})$ appears in the new type assumption, hence (5.3.3) is satisfied with the annotation $y^{\tilde{x}}$. By induction hypothesis, $J_1\sigma$, $J_2\sigma$ satisfy (5.3.4). In particular, this is the case for the derived type t, which is copied to the conclusion $J\sigma$, which hence also fulfills (5.3.4). •

Generalization We exploit the Proposition 5.3.2 to implement the predicate `gen`. It takes five parameters: The bound name which is introduced for quantified variables, the last used index in the vector $\tilde{\alpha}$ and the type to be generalized. Parameters 4 and 5 are the result, consisting of the new last used index and the generalized type.

Let us consider the straightforward cases of function-, boolean and integer types first. Note how the last used index is handed on between the premises of `gen_fun` to ensure that each index is used at most once. Note also that these rules cannot be applied arbitrarily, precisely because they would instantiate remaining unknowns in the types; Section 5.3.2.4 gives the necessary tactics for applying them in correct order.

```
gen_fun  "[| gen(z,i,s,i',s'); gen(z,i',t,i'',t') |]
          ==> gen(z,i,s -> t, i'', s' -> t')"
gen_int  "gen(z,i,int,i,int)"
gen_bool "gen(z,i,bool,i,bool)"
```

The main case arises when `gen` meets an unknown u, applied to various bound names $\tilde{z}$ (for unification [Nip93]) in a goal

$$\texttt{gen}(\texttt{z}, i, u(\tilde{z}), i', u'(\tilde{z}))$$

where i' and u' are variables to be instantiated with the result. Using Proposition 5.3.2, `gen` decides whether to generalize the unknown u by setting $u' := \texttt{z}$, or whether to leave the unknown as a monomorphic type variable, setting $u' := u$.

The first rule instantiates $u(\tilde{z})$ with a new term $\texttt{nth}(\texttt{z}, i)$, that is, it generalizes the unknown. By definition of higher-order unification, this rule succeeds only if $\texttt{z} \in \tilde{z}$. By the Proposition, this condition coincides with the condition $\tilde{\alpha} = \mathbf{FV}(s) \setminus \mathbf{FV}(\Gamma)$ in (let). The new last used index must be incremented if the rule is applied.

```
  gen_gen_n "gen(z,i,nth(z,i),  ni(i), nth(z,i))"
```

The second time that we try to generalize an unknown, this rule fails because the index i has changed in the second argument, while the index of the first generalization is found in the third argument. Hence, we need yet another rule which just retains the old index.

```
  gen_gen_o "gen(z,i,nth(z,i'), i, nth(z,i'))"
```

If these two rules fail, then `z` $\notin \tilde{z}$ and u must not be generalized, but left as a type unknown. To check that indeed an unknown is present, we tentatively instantiate it with a constructor `TYV` that is used nowhere else. (Note that any other occurrence of u will be replaced simultaneously, hence the result variables for all of these occurrences will be unified. The constructor is removed later as in instantiation.)

```
  gen_var "gen(z,i,TYV(a),i,a)"
```

5.3.2.4 Proof Search

Proof search for type inference is essentially straightforward, since the type system is syntax directed, that is there is exactly one typing rule for each language construct. Therefore, all our tactics will resolve the first goal in the proof state, deviating from the Isabelle convention to make the goal to be treated a parameter of the tactic. The only intricate point is the predicate `gen`, which must make sure that the rules are applied in the appropriate order, as captured by the following tactic. Note that tactical `FIRST` does not create alternatives for backtracking: The first successful tactic is chosen. After the `gen` predicates are resolved (tactical `REPEAT1` fails if the repeated tactic is not applied at all), the auxiliary `TYV` constructors are removed.

```
val gen_tac  =
  (REPEAT1 (FIRST [ (resolve_tac [ gen_gen_n ] 1),
                    (resolve_tac [ gen_gen_o ] 1),
                    (resolve_tac [ gen_var   ] 1),
                    (resolve_tac [ gen_fun   ] 1),
                    (resolve_tac [ gen_int   ] 1),
                    (resolve_tac [ gen_bool  ] 1) ] ))
  THEN (rewrite_tac [ remove_TYV ]);
```

Instantiation with `inst` works in the same manner. For *let*-bound variables, their type schemes must be instantiated. This happens in three steps: First, using rule `var`, a goal with predicate `:*` is created, which is resolved against the assumptions to obtain the type scheme. We then know that the second goal of `var` is still pending and resolve it with `inst_tac`.

```
val var_tac  =
  ((resolve_tac [ var ] 1)
  THEN (assume_tac 1)
  THEN inst_tac);
```

Finally, the complete tactic for type checking tries the different possible cases of expressions in turn: `assume_tac` looks up λ-bound identifiers in the assumptions, `var_tac` treats *let*-bound variables, `cnst_tac` treats literals, if, let and application and λ abstraction. `land_tac` resolves the auxiliary `&`.

```
val ty_tac =
  REPEAT1
   (FIRST [
     (assume_tac 1),
     var_tac,
     cnst_tac,
     land_tac,
     gen_tac
   ]);
```

To go one step further, we define a function to type check a given expression. Isabelle prints the result when the proof is complete.

```
  fun tychk e = prove_goal MiniML.thy e (fn _ => [ ty_tac ]);
```

5.3.2.5 Example Typings

We now give some example applications with the answers generated by Isabelle.[16] The first example is restricted to the simply typed λ-calculus. It essentially adds `1` and `2`, returning an integer.

```
tychk "(lam f . f ‘ 1) ‘ (lam x . (x + 2)) : ?t";
val it = "(lam f. f ‘ 1.0) ‘ (lamx. x + ?2.0) : int" : thm
```

The second example show that the type of λ-arguments is inferred by unification.

```
tychk "lam x . x + 1 : ?t";
val it = "lam x. x + 1.0 : int -> int" : thm
```

The first example involving polymorphic *let* type-checks the identity function and applies the function to itself. The type scheme inferred for `f` should be read as $\forall\alpha_0.\alpha_0 \rightarrow \alpha_0$.

```
tychk "let f : ?tf = lam x . x in f ‘ f : ?t";
val it =
  "let f : ALL z. NTH(z, 0) -> NTH(z, 0)  = lam x. x in f ‘ f :
   ?s’ -> ?s’" : thm
```

The next example involves a vector of generalized variables for the type scheme of $\lambda x.\lambda y.x$. That type scheme is $\forall\alpha_0,\alpha_1.\alpha_0 \rightarrow \alpha_1 \rightarrow \alpha_0$.

```
tychk "let fst : ?tfst = lam x. lam y. x in fst ‘ fst : ?t";
val it =
  "let fst : ALL z. NTH(z, 0) -> NTH(z, 1) -> NTH(z, 0)  = lam x y. x
   in fst ‘ fst : ?s’ -> ?s’a -> ?s’b -> ?s’a" : thm
```

[16]The notation `?u` denotes an unknown u in Isabelle. `val it =` indicates the last derived fact.

Finally, we consider the expression

$$\lambda b.\lambda x.\texttt{let}\ f = \lambda y.\texttt{if}\ b\ \texttt{then}\ x\ \texttt{else}\ y\ \texttt{in}\ f$$

The type of y cannot be generalized, because by the rule for `if`, it must be equal to the type of x, which is bound outside of the `let`. Our implementation correspondingly refuses the generalization of `?s'`, because in the application of `tyif`, the types of x and y are unified. At that point, the type of y looses its annotation with the bound name `z`.

```
tychk "lam b x. let f : ?tyf = lam y. if b then x else y in f : ?t";
val it =
  "lam b x.
      let f : ALL z. ?s' -> ?s'  = lam y. if b then x else y in f :
   bool -> ?s' -> ?s' -> ?s'" : thm
```

5.3.2.6 Conclusion

The theory `MiniML` developed in this section shows that it is feasible to use a logical framework like ISABELLE to encode and execute type checkers. However, I wish to stress two observations that justify an independent development like TCG:

- ISABELLE's parser generator treats only grammars that extend the meta-logic. This works well for mathematical notations, which may also be modified to suit the framework. A parser generator for programming languages should not be prejudiced in this way. For instance, it was necessary to introduce the auxiliary operator ` for application, because the usual application by juxtaposition was already used in the `Pure` function application. (Strictly speaking, `Pure` uses $f(a_1 \mathinner{\ldotp\ldotp} a_n)$ which creates ambiguities nevertheless. The ambiguities can be resolved by type annotations, since only type correct parses are accepted, but this is cumbersome.)
- The use of higher-order abstract syntax facilitates the encoding of bound names. However, it fixes the form of typing rules and moreover introduces a significant problem of type inference for polymorphic *let*, which could only be solved by a novel development.
- The solution for polymorphic let in this section bears similarities with TCG's notions of *inner* and *outer* variables (Sections 1.2.3.5, 2.3, 2.4.1) which also characterize the occurrence of variables in the proof tree relative to the application of the (let) inference rule. In terms of Section 1.2.2, TCG's formulation is in the style of natural deduction and reasons about entire deductions, while the ISABELLE theory, driven by the representation of goals, is in sequent style.

Chapter 6

Conclusion

6.1 Summary

I have presented an approach to obtaining executable type checkers from descriptions of type systems, the Type Check Generator (TCG). The approach is based on the key abstraction of type-checking-as-proof-search: Type systems are formalized as sets of rules, and an interpreter constructs proofs for a given typing judgement.

Suitable notions of *rule* and *proof* (Chapter 2) have been derived by analyzing common typing rules and the proof theory of type systems and logics (Section 1.2). The most characteristic feature of TCG's proofs is subproof extraction (Section 1.2.3.5), which converts a sub-tree of the proof tree to a TCG rule. Subproof extraction captures type inference for polymorphic *let* (Section 1.2.3.5, 4.1.3.3) and exhibits close meta-theoretical connections to the soundness proofs for that construct (Section 2.4.1.2). It also enables direct renderings of typing rules that otherwise would have to be programmed, rather than declared (Chapter 4).

The TCG proofs and rules permit an interpreter based on backward, Prolog-style resolution of judgments (Section 2.4, 3.3.4). The interpretation process is factored into atomic steps, which are functions from proofs to proofs and which maintain the invariants on the structure of proofs. Proof construction is then a depth-first backtracking search for complete proofs under repeated application of atomic derivation steps.

TCG is complementary to constraint-based approaches to type checking (Section 5.2). These approaches assume that a given type checker traverses the parse tree of the program and generates a set of constraints over types that must be satisfied for the program to be well-typed. Constraint-based abstractions thus capture the treatment of constraints once they have been generated, while TCG captures the process of traversing the parse tree, abstracting over the programming language. The integration of TCG with constraint handling formalisms is a matter of future research (Section 6.2.3). TCG differs from most previous, Prolog- or rewriting-based, approaches to describing type checkers (Section 5.1) in that its formalism is tailored to the specific format of typing rules to permit a declarative description of the type system.

I have given a structure-sharing implementation of TCG (Chapter 3), where proofs share their data structure with alternatives and predecessors and each derivation step records only the pointer modifications necessary to construct the result proof from the input proof. This strategy is particularly well-suited for depth-first search, where the incremental modifications between steps are small. It generalizes the Prolog trail of substitutions [Bru82, AK91, VR94] to maintenance of the entire proof structure. The implementation directly represents the proofs, rules and inference steps from Chapter 2. An inspection mechanism with a graphical user interface permits the user to observe the type checker's operation in detail. Although due to structure sharing the efficiency is reasonable, a faster interpreter will be necessary for checking longer programs (Section 6.2.1).

Chapter 4 contains extensive applications of Tcg to functional, imperative and object-oriented languages, treating their salient features, including ML-style type inference. In most cases, the type requirements for a language construct can be captured directly in single Tcg rule — unlike many earlier approaches (Section 5.1) there is no need for "programming" the type checker by rules. Since the typing rules are thus arranged around language constructs, they are largely orthogonal and exhibit a high potential for re-use, which I exploit in a library of Tcg rules. That library also forms the imperative kernel of Saga, a novel language design aiming at generic programming (Section 4.5). It permits the statement of generic algorithms with requirements on their parameter types [MS89, MS94, Sch96a, Aus98]. The requirements arising from an algorithm call are checked locally against the context of the call, such that generation of concrete instances never fails if the type checker accepts the program.

6.2 Future Directions

6.2.1 Implementation

Term Indexing The current implementation (Section 3.3.2) filters rules before application by their conclusions' top-level symbols. Since type assumptions are represented by rules and they all share the top-level symbol (which marks them as type assumptions), this filter is clearly insufficient for type checking larger programs. A suitable extension are discrimination tree indexes [Gra96, Section 6.1]. However, two adaptations are required: First, the positions within terms that should contribute to discrimination must be declared by the user. For example, indexing a type assumption $x \mapsto x' : t$ (Section 4.2.1.1) in the internal name x' and type t only consumes memory, it does not improve the filter, since already x alone is unique (apart from overloading (Section 4.3.4), where it is desirable to find all alternatives). The choice of a discrimination tree entails that nested name spaces are mapped to nested sub-indexes in much the same way that maps represent environments in hand-written type checkers. As a second adaptation, the structure of indexes must be shared between judgments to a large degree, otherwise the positive effect on the run-time is canceled by a negative effect on memory consumption. In the batch interpreter to be sketched in the next paragraph, only one context will be accessible at any point in time, hence structure-sharing can be implemented analogously to that of proofs (Section 3.3.3.1).

Batch Interpreter The implementation of Chapter 3 is designed to represent the definitions from Chapter 2 directly to demonstrate their adequacy for describing type checkers and to make the effects of TCG rules visible in detail (Section 3.5). However, it maintains solved, intermediate judgments that are not accessed in the subsequent proof search. For a batch interpreter, it is sufficient to build proof construction around a recursive function `solve` that resolves a single judgement and returns the necessary substitution and the exports and deferred judgments of the constructed sub-proof (Section 2.5.1). The existing implementation should also be used to identify frequent special cases of rules.

6.2.2 Extensions

Bound Names in Terms Since lexically bound variables are ubiquitous in typed languages, it is desirable to provide bound variables in TCG terms for their representation [Pfe01] (also Section 5.3.1). Higher-order pattern unification [Nip93, Mil92, Dow01] has already been used in conjunction with logic programming [Mil91, NM98, Wic99] and a similar synthesis is possible with TCG.

Proof Selection Type checking for realistic languages often involves disambiguation rules that are not strictly part of the type system, but choose between several type-correct interpretations of the program source. The most frequent cases are overload resolution based on a "best match" selection (C^{++}, Java) and selection of coercions. Such decisions appear orthogonal to the questions considered in this thesis. We have identified TCG proofs with type-correct interpretations of the program source, hence *any* constructible proof is a valid answer to the typing question. The disambiguation rules then serve to *select* among the valid answers.

One *a priori* selection mechanism is already implemented by the proof grammars (Section 3.1.4). They deliberately restrict the set of proofs that can be constructed by overlaying the proof tree with a regular tree grammar [$CDG^{+}99$]. An *a posteriori* selection of completed proofs should be connected with the *branch* goals (Sections 2.4.5, 3.1.3), since this is the only place where collections of proofs are accessible in the construction process. A predicate could be added to the *expand* instruction to select the desired proof. However, it remains to be investigated which characteristics of proofs should be accessible to the predicate and in what formalism the predicate should be written. A first solution is a general ML function that accesses the internal proof structures through a custom interface.

Types for Tcg For a formal interpretation of TCG proofs as type derivations, it is necessary to derive the form of terms that are substituted for variables in rules [Gas01, Chapter 2][Pfe01, Section 2]. The same information would be desirable to check statically for mistakes that lead to judgments without matching rules; furthermore, groundness of terms in specific positions is often required for the termination or efficiency of the type checkers. These questions have been well-investigated in type inference for Prolog [HJ90, FSVY91, Hei92, JB92, GdW92, DZ92, GdW94, AL94, TTD97, DTT97, CP98]. The main concern, the propagation of terms by unification, is shared with TCG's model of execution. Hence, with adaptations for approximating the contexts of judgments, the developed techniques are applicable.

6.2.3 Constraint Handling

Since an extensive set of constraint domains and corresponding constraint solvers is available (Section 5.2) it would be desirable to make their expressive power accessible to TCG type checkers. This would make a wide variety of type systems amenable for treatment, including general subtyping constraints [Mit91, AW93, EST95a, Pot01], and dependent types over integer domains [XP99]. Also non-syntactic type equality, for example modulo β-reduction [ML84, CH88, Bar91][1] or with cyclic dependencies (recursive types) [AC93], then becomes an option.

The linking point with the existing TCG proofs (Chapter 2) are the deferred judgments (Section 1.2.3.6), which represent proof obligations that cannot be solved sensibly by a backtracking search. Operationally, the desired extension is straightforward. The special instructions *expand* and *cut* (Section 3.1) are complemented with an instruction $\mathit{solve}(i_1 \mathinner{..} i_n)$, which is resolved by handing the deferred judgments in the sub-proofs for premises $i_1 \mathinner{..} i_n$ to a constraint solver. The result is a proof delta (Section 3.3.3.1) that is applied to the input proof to obtain the result of the derivation step. The proof delta includes a substitution for variables, addition of new judgments and solving of existing judgments. Hence, for example a normalization step in the sense of [Sul00, Section 4.2] can be expressed. Choosing constraint handling rules (CHR [Frü98]) as the formalism to specify constraints links TCG to recent work on type checking by constraint solving [SS01, Sul01, AF02, AF04].

However, the result of constraint solving is not necessarily a proof in the sense of Section 2.3 and a corresponding extension of the notion of proofs is required. A straightforward solution [Jon94, Sul00, EST95a, Pot01] is to augment judgments with a set of constraints P. The defer step (Section 2.4.4) would be complemented with an axiom

$$\frac{}{P \cup \{d\} \mid \Gamma \vdash d} \text{(Constraint)}$$

The set P could be implemented as a structure-shared, mutable list (Section 3.3.3.1) that is attached directly to the *solve* instruction introduced above. Since the context Γ is no longer available in resolving $d \in P$, (Constraint) can be applied only when the context becomes immaterial. Hence, the existing mechanism for deferring goals temporarily (Section 3.1.4) is still needed. For example, Haskell's type class constraints [WB89, Jon94] must be simplified under the instance declarations in the context of the judgement [Jon99], which depends on the static environment of the type-checked expression.

[1]Since β-reduction is strongly normalizing in the presence of kinds, the equality constraints could be also solved with the current TCG by programming β-reduction explicitly [CH88, Section 6.1].

Appendix A

Notation and Trees

Notation

Symbol	Meaning
X^*	the sequences over X
ε	empty sequence
$px, p \cdot x, p \cdot q, pq$	concatenation
$p \leq q \quad (p < q)$	p is a (proper) prefix of q
$x_1 \mathinner{..} x_n$	the elements of a set or sequence
$\tilde{x}$	— ” — where n is irrelevant
$\langle x_1 \mathinner{..} x_n \rangle$	a sequence
$\equiv$	syntactic equivalence
$=$	equality
$t[s_1/x_1 \mathinner{..} s_n/x_n]$	simultaneous (capture-avoiding) substitution of s_i for x_i.
$\mathbb{N}$	natural numbers including 0
$\mathbb{N}_+$	positive natural numbers
$\sqcup$	least upper bound
$\sqcap$	greatest lower bound
$\rightarrow$	function space
$\overset{\text{fin}}{\rightarrow}$	function with finite domain
$\downarrow$	“undefined” (or $\perp$) for partial functions
$f\vert_D$	function f restricted to D
$f\vert_{\complement D}$	function f restricted to the complement of D
$\complement X$	set complement
@	application if infix symbol is required

Tree Notation	
$t[p]$	tree node in t at path p
$t\{p\}$	subtree of t rooted at path p
$t\backslash_p$	tree t pruned at p ($:= \{d \mapsto l \mid d \mapsto l \in t, p \not\leq d\}$)
Defined Concepts	
IV	inner variables (Section 2.3)
OV	outer variables (Section 2.4.6)
Naming Conventions	
$\alpha, \beta, \gamma, \delta$	type variables
σ, τ, ϕ	substitutions
ρ	renaming
Γ	type assumptions
$s, t, r, \ldots$	types and type schemes
e, f, g	expressions
x, y, z	identifiers (variables) in expressions

Trees

Definition. (Tree Domain [Gal86, Section 2.2.1])
A *tree domain* D is a nonempty subset of strings in $\mathbb{N}_+^*$ satisfying the conditions:

1. For each $u \in D$, every prefix of u is also in D.
2. For each $u \in D$, for every $i \in \mathbb{N}_+$, if $ui \in D$ then, for every j, $1 \leq j \leq i$, uj is also in D.

Definition. (Trees [Gal86, Section 2.2.2])
Given a set Σ of labels, a Σ-tree (for short, a *tree*) is a total function $t : D \to \Sigma$, where D is a tree domain.

Bibliography

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping Recursive Types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

[AC96] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, 1996.

[ACCL91] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(3):375–416, October 1991.

[AF02] Sandra Alves and Mário Florido. Type inference using constraint handling rules. *Electronic Notes in Theoretical Computer Science*, 64:17, 2002.

[AF04] Sandra Alves and Mário Florido. Type inference for programming languages: A constraint logic programming approach. Technical Report DCC-2004-5, Departamento de Ciência de Computadores, Universidade do Porto, 2004. `http://www.ncc.up.pt/fcup/DCC/Pubs/treports.html`.

[AFFS98] Alexander Aiken, Manuel Fähndrich, Jeffrey S. Foster, and Zhendong Su. A Toolkit for Constructing Type- and Constraint-Based Program Analyses. In Leroy [Ler98], pages 78–96.

[AFM97] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding Type Parameterization to the Java Language. In *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '97)*, volume 32 (10) of *SIGPLAN Notices*, pages 49–65, Atlanta, Georgia, October 1997. ACM.

[AHS86] Aho, Hopcroft, and Sethi. *Compilers – Principles, Tools, Techniques.* Addison-Wesley, 1986.

[AK91] Hassan Aït-Kaci. *Warren's abstract machine: A tutorial reconstruction.* MIT Press, 1991.

[AL94] Alexander Aiken and T. K. Lakshman. Directional Type Checking of Logic Programs. Technical Report UCB/CSD 94-791, Computer Science Division (EECS), University of California, Berkeley, California, 1994.

[And92] James H. Andrews. *Logic programming : operational semantics and proof theory.* Distinguished dissertations in computer science. Cambridge University Press, Cambridge, 1992.

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML.* Cambrigde University Press, Cambridge, United Kingdom, 1998.

[Aug98] L. Augustsson. Cayenne – a language with dependent types. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 239–250, Baltimore, Maryland, United States, 1998. ACM SIGPLAN.

[Aus98] Matthew H. Austern. *Generic Programming and the STL — using and extending the C^{++} Standard Template Library.* Addison-Wesley, 1998.

[AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Conference on Functional Programming Languages and Computer Architecture*, pages 31–41. ACM press, 1993.

[AWL94] Alexander Aiken, Edward L. Wimmer, and T.K. Lakshman. Soft Typing with Conditional Types. In *21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages '94*, pages 163–173, Portland,Oregon, 1994. ACM.

[Bak82] T.P. Baker. A One-Pass Algorithm for Overload Resolution in Ada. *ACM Transactions on Programming Languages*, 4(4):601–614, 1982.

[Bar84] H.P. Barendregt. *The Lambda Calculus–Its Syntax and Semantics.* North-Holland, Amsterdam, 2nd edition, 1984.

[Bar91] Henk Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(1):125–154, January 1991.

[Bar95] John Barnes. *Programming in Ada 95.* International computer science series. Addison-Wesley, Wokingham, 1995.

[BCD^{+}89] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: the system. *ACM SIGPLAN Notices*, 24(2):14–24, February 1989.

[BCK^{+}01] Gilad Bracha, Norman Cohen, Christian Kemper, Steve Marx, Martin Odersky, Sven-Eric Panitz, David Stoutamire, Kresten Thorup, and Philip Wadler. Adding generics to the java programming language: Participant draft specification. Sun preliminary specification, April 2001.

[BG00] Henk Barendregt and Silvia Ghilezan. Lambda terms for natural deduction, sequent calculus and cut elimination. *Journal of Functional Programming*, 10(1):121–134, 2000.

[BH97] Michael Brandt and Fritz Henglein. Coinductive Axiomatization of Recursive Type Equality and Subtyping. In Philippe de Groote, editor, *Third International Conference on Typed Lambda Calculi and Applications, TLCA '97*, volume 1210 of *Lecture Notes in Computer Science*, pages 63–81, Nancy,France, 2–4April 1997. Springer.

[BHKO02] M.G.J. van den Brand, J. Heering, P. Klint, and P.A. Olivier. Compiling language definitions: The ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, July 2002.

[BM72] R.S. Boyer and J S. Moore. The sharing of structure in theorem-proving programs. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 7, pages 101–116. Edinburgh University Press, 1972.

[BN98] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, Cambridge, 1998.

[Boo91] Grady Booch. *Object oriented design with applications.* Benjamin/Cummings Publishing Company, Redwood City, CA, 1991.

[Bru82] Maurice Bruynooghe. The memory management of Prolog implementations. In Clark and Tärnlund [CT82].

[BS86] Rolf Bahlke and Gregor Snelting. The PSG system: From formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, October 1986.

[BS01] Franz Baader and Wayne Snyder. Unification theory. In Robinson and Voronkov [RV01], chapter 8, pages 445–533.

[BSG03] Kim B. Bruce, Angela Schuett, and Robert von Gent. PolyTOIL:A Type-Safe Polymorphic Object-Oriented Language. *ACM TOPLAS*, 25(2), March 2003.

[BTCGS91] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as Implicit Coercion. *Information and Computation*, 93:172–221, 1991.

[Car87] Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2), April 1987.

[Car93] Luca Cardelli. An implementation of $F_{<:}$. Technical Report 97, DEC Systems Research Center,Palo Alto, February 1993.

[Cas95] Giuseppe Castagna. Covariance and Contravariance – Conflict without a Cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, May 1995.

[CDDK86] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A Simple Applicative Language: Mini-ML. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 13–26, Cambridge, Massachusetts, USA, August 1986. ACM.

[CDG+99] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree Automata – Techniques and Applications. `http://www.grappa.univ-lille3.fr/tata/`, October 1999.

[CH88] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, February 1988.

[CHC89] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 125 –135, San Francisco, California, United States, 1989. ACM.

[CHW98] James Coplien, Daniel Hoffman, and David Weiss. Commonality and variability in software engineering. *IEEE Software*, pages 37–45, November 1998.

[CM94] Luca Cardelli and John C. Mitchell. Operations on records. In Gunter and Mitchell [GM94], pages 295–350. Also appear as: SRC Research Report 48, 1989, Digital Equipment Corporation; Mathematical Structures in Computer Science, vol. 1.

[Cop98] James O. Coplien. *Multi-Paradigm Design for C^{++}*. Addison-Wesley, 1998.

[Cor82] G.V. Cormack. An Algorithm for the Selection of Overloaded Functions in Ada. *ACM SIGPLAN Notices*, 16(2):48–51, 1982.

[Cou83] Bruno Courcelle. Fundamental Properties of Infinite Trees. *Theoretical Computer Science*, 25:95–169, 1983.

[CP98] Witold Charatonik and Andreas Podelski. Directional Type Inference for Logic Programs. In *Static Analysis, 5th International Symposium, SAS '98*, volume 1503 of *Lecture Notes in Computer Science*, pages 278–294, Pisa, Italy, September 1998. Springer-Verlag.

[CT82] Keith L. Clark and Sten-Ake Tärnlund, editors. *Logic programming.* A.P.I.C. studies in data processing. Academic Press, London, 1982.

[CW85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4):471–520, December 1985.

[Des84] Thierry Despeyroux. Executable specification of static semantics. In *Semantics of Data Types, International Symposium*, number 173 in Lecture Notes in Computer Science, pages 215–231, Sophia-Antipolis, France, June 1984. Springer-Verlag.

[Deu91] A. van Deursen. An algebraic specification for the static semantics of pascal. Technical Report CS-R9129, Centrum voor Wiskunde en Informatica (CWI), 1991.

[Deu96] Arie van Deursen. Appendix: “Pascal should be type checked; A readable approach to formalizing the static semantics of Pascal” of “The static semantic of Pascal”. In Arie van Deursen, Jan Heering, and Paul Klint, editors, *Language Prototyping: An algebraic specification approach*, AMAST Series in Computing. World Scientific Publishing Co., 1996.

[DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 207–212, Albuquerque, New Mexico, January 1982.

[Dow01] Gilles Dowek. Higher-order unification and matching. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 16, pages 1011–1062. Elsevier Science, Amsterdam, The Netherlands, 2001.

[DP91] Scott Dietzen and Frank Pfenning. A declarative alternative to ”assert” in logic programming. In Vijay A. Saraswat and Kazunori Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium*, pages 372–386, San Diego, California, USA, Oct 28 - Nov 1, 1991. MIT Press.

[DTT97] Philippe Devienne, Sophie Tison, and Jean-Marc Talbot. Solving classes of set constraints with tree automata. In Gert Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming (CP 97)*, volume 1330 of *Lecture Notes in Computer Science*, pages 62–76, Linz, Austria, October 29–November 1 1997. Springer-Verlag.

[DZ92] Philip W. Dart and Justin Zobel. A Regular Type Language for Logic Programs. In Frank Pfenning, editor, *Types in Logic Programming*, Logic Programming, chapter 5. MIT Press, Cambridge, Massachusetts, 1992.

[Ede85] Elmar Eder. Properties of substitutions and unifications. *Journal of Symbolic Computation*, 1:31–46, 1985.

[EN94] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of database systems.* Benjamin/Cummings, Redwood City, CA, 2nd edition, 1994.

[EST95a] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. *Electronic Notes in Theoretical Computer Science*, 1, 1995.

[EST95b] Jonathan Eifrig, Scott F. Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 169–184, 1995.

[Fel93] Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 1993.

[FF96] Cormac Flanagan and Matthias Felleisen. Modular and polymorphic set-based analysis. Technical Report Rice COMP TR96-266, Department of Computer Science Rice University, P.O. Box 1892, Houston, TX 77251-1892, November 1996.

[FF99] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. *ACM Transactions on Programming Languages and Systems*, 21(2):370–416, March 1999.

[FKS94] Zsuzsa Farkas, Péter Köves, and Péter Szeredi. MProlog: an implementation overview. In Evan Tick and Giancarlo Succi, editors, *Implementations of Logic Programming Systems*, pages 103–117, Boston 1994, 1994. Kluwer Academic Publishers.

[Frü98] Thom Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.

[FSVY91] Thom Frühwirth, Ehud Shapiro, Moshe Y. Vardi, and Eyal Yardeni. Logic programs as types for logic programs. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 300–309, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[Gal86] Jean H. Gallier. *Logic for Computer Science – Foundations of Automatic Theorem Proving.* Harper & Row Publishers, 1986.

[Gas01] Holger Gast. Generic Programming with Views: Type- and Class-inference with Polymorphic Subsumption by Resolution Theorem Proving. Technical Report WSI-2001-17, Wilhelm-Schickard Institut,Universität Tübingen, November 2001.

[gcj04] GNU Compiler for the Java programming language. `http://gcc.gnu.org/java`, April 2004.

[GdW92] J.P. Gallagher and D.A. de Waal. Regular approximations of logic programs and their uses. Technical report, Department of Computer Science, University of Bristol, Queen's Building, University Walk, Bristol BS8 1TR, U.K., March 1992.

[GdW94] J.P. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. In Pascal Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613. The MIT Press, 1994.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schliessen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in [Sza69, pages 68–131].

[Ghi99] Silvia Ghilezan. Natural deduction and sequent typed lambda calculus. *Novi Sad Journal of Mathematics*, 29(3):207–218, 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Professional Computing Series. Addison-Wesley, 1995.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[GJS00] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 2nd edition, 2000.

[GM94] C. Gunter and J.C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design.* MIT Press, Cambridge, MA, 1994.

[GMW79] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1979.

[GR80] Harald Ganzinger and Knut Ripken. Operator Identification in ADA: Formal Specification, Complexity, and Concrete Implementation. *ACM SIGPLAN Notices*, 15(2):30–42, February 1980.

[Gra96] Peter Graf. *Term Indexing*, volume 1053 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, 1996.

[GS02] Holger Gast and Christoph Schwarzweller. Local Checks for Semantic Requirements of Generic Algorithms. (unpublished manusript), September 2002.

[GSL97] H. Gast, S. Schupp, and R. Loos. Completing the Compilation of SuchThat v0.7. Technical Report 97-12, Rensselaer Polytechnic Institute, December 1997.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types.* Number 7 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[Han98] John Hannan. Program analysis in lambda prolog (tutorial slides). Presented at PLILP '98, Pisa, Italy, September 1998. available from `http://www.cse.psu.edu/~hannan/papers.html`.

[Hei92] Nevin Heintze. *Set Based Program Analysis.* PhD thesis, School for Computer Science, Computer Science Division, Carnegie Mellon University, Pittsburgh, USA, 1992.

[Hen89a] P.R.H. Hendriks. Typechecking Mini-ML. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Frontier Series, chapter 7. ACM Press, 1989.

[Hen89b] Fritz Henglein. *Polymorphic Type Inference and Semi-Unification.* PhD thesis, Courant Institute of Mathematical Sciences, New York, NY, USA, April 1989. Appeared as NYU/Courant Institutte of Mathematical Sciences Technical Report 443, May 1989.

[Hen93] Fritz Henglein. Type Inference with Polymorphic Recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[Her92] Helmut Herold. *Lex und Yacc : lexikalische u. syntaktische Analyse.* UNIX und seine Werkzeuge. Addison-Wesley, Bonn, 1992.

[HHJW96] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type Classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, 1996.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[Hin69] Roger Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[HJ90] Nevin Heintze and Joxan Jaffar. A Finite Presentation Theorem for Approximating Logic Programs (Extended Abstract). In *Seventeenth Annual ACM Symposium onn Principles of Programming Languages*, pages 197–209, San Francisco, California, January 17–19, 1990. ACM Press, New York.

[HK00] Jan Heering and Paul Klint. Semantics of programming languages: A tool-oriented approach. *SIGPLAN Notices*, 35(3):39–48, March 2000.

[HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 123–137, Portland, OR, January 1994.

[Hoa72] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(1):271–281, 1972.

[How80] W.A. Howard. The formulae-as-types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, London, 1980.

[HPM$^+$02] Pedro Henriques, Maria Varanda Pereira, Marjan Mernik, Mitja Lenic, Enis Avdicausevic, and Viljem Zumer. Automatic generation of language-based tools. *Electronic Notes in Theoretical Computer Science*, 65(3), 2002.

[HR95] Fritz Henglein and Jakob Rehof. Safe polymorphic type inference for a dynamically typed language: Translating Scheme to ML. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture (FPCA), La Jolla, California.* ACM Press, 1995.

[Hue75] G.P. Huet. A Unification Algorithm for Typed λ-Calculus. *Theoretical Computer Science*, 1(1):27–57, 1975.

[ISO98] ISO. International Organization for Standardization: Programming languages – C++. Number ISO/IEC 14882, 1998.

[JB92] G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2&3):205–258, 1992.

[Je99] Jones and Hughes (eds.). Report on the Programming Language Haskell 98 — A Non-strict,Purely Functional Language. `http://www.haskell.org/definition/haskell98-report.ps.gz`, February 1999.

[JJ97] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482, 1997.

[Jon87] Simon L. Peyton Jones. *The Implementation of functional Programming Languages.* Series in Computer Science. Prentice/Hall, May 1987.

[Jon94] Mark P. Jones. A theory of qualified types. *Science of Computer Programming*, 22(3):231–256, June 1994.

[Jon95] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–35, January 1995.

[Jon99] Mark P. Jones. Typing Haskell in Haskell. In Erik Meijer, editor, *Proceedings of the 1999 Haskell Workshop*, number UU-CS-1999-28 in Technical Report, pages 1–14, Paris, France, October 1999. University of Utrecht, Institute of Information and Computing Sciences. Also available from `http://www.cse.ogi.edu/~mpj/thih/`.

[JP99] Trevor Jim and Jens Palsberg. Type inference in systems of recursive types with subtyping. available from `http://www.cs.purdue.edu/homes/palsberg/publications.html`, June 1999.

[JS92] Richard D. Jenks and Robert S. Sutor. *AXIOM : the sientific computation system.* Springer-Verlag, New York, 1992.

[JW85] Kathleen Jensen and Niklaus Wirth. *Pascal – User Manual and Report.* Springer Verlag, New York, 3rd edition, 1985.

[Kah87] Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *4th Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39, Passau, Germany, February 1987. Springer.

[Knu68] D.E. Knuth. Semantics of context-free languages. *Math. Syst. Theory*, 2(2):127–145, June 1968.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language.* Prentice Hall software series. Prentice Hall, Englewood Cliffs, NJ, 2nd, ANSI C edition, 1988.

[KTU93] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. *Information and Computation*, 102:83–101, 1993.

[Ler95] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 142–153, San Francisco, California, January 1995. ACM Press.

[Ler98] Xavier Leroy, editor. *Types in compilation : second international workshop (TIC98)*, volume 1473 of *Lecture Notes in Computer Science*, Kyoto, Japan. March 1998, 1998. Springer.

[Ler00] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, May 2000.

[Li98] Xining Li. A new term representation method for Prolog. *Journal of Logic Programming*, 34(1):43–58, 1998.

[Lia97] Chuck Liang. Let-polymorphism and eager type schemes. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France, April 14-18, 1997, Proceedings*, volume 1214 of *Lecture Notes in Computer Science*, pages 49–501. Springer, 1997.

[Lia02] Chuck Liang. Compiler construction in higher order logic programming. In Shriram Krishnamurthi and C. R. Ramakrishnan, editors, *Practical Aspects of Declarative Languages, 4th International Symposium, PADL 2002*, volume 2257 of *Lecture Notes in Computer Science*, pages 47–63, Portland, OR, USA, 2002. Springer.

[Lip96] Stanley B. Lippman. *Inside the C++ object model.* Addison-Wesley, 1996.

[Lit98] Vassily Litvinov. Contraint-based polymorphism in cecil: towards a practical and static type system. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 388 – 411, Vancouver, British Columbia, Canada, 1998. ACM Press.

[LP03] Michael Y. Levin and Benjamin C. Pierce. TinkerType: A language for playing with formal systems. *Journal of Functional Programming*, 13(2), March 2003.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, November 1994.

[McC94] William W. McCune. Otter 3.0 reference manual and guide. Technical Report ANL 94/6, Argonne National Laboratory, January 1994.

[Mel82] C.S. Mellish. An alternative to structure sharing in the implemenatation of a Prolog interpreter. In Clark and Tärnlund [CT82].

[Mil78] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[Mil92] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14(4):321–358, October 1992.

[Mit90] J.C. Mitchell. Type systems for programming languages. In Jan van Leeuwen, editor, *Handbook of Theorectical Computer Science*, volume B – Formal Models and Semantics, chapter 8, pages 367–458. Elsevier and MIT Press, 1990.

[Mit91] John C. Mitchell. Type Inference With Simple Subtypes. *Journal of Functional Programming*, 1(3):245–285, July 1991.

[ML84] Per Martin-Löf. *Intuitionistic type theory.* Bibliopolis, 1984.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[Mor73] James H. Morris. Types are not sets. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 120–124, Boston, Massachusetts, 1973.

[MP88] John C. Mitchell and Gordon D. Plotkin. Abstract Types Have Existential Type. *ACM Transactions on Programming Languages and Systems*, 10(8):470–502, July 1988.

[MPS86] David McQueen, Gordon Plotkin, and Ravi Sethi. An Ideal Model for Recursive Polymorphic Types. *Information and Control*, 71(1/2):95–130, 1986.

[MS89] Musser and Stepanov. Generic programming. In *Proceedings of the ACM SIGSAM International Symposium on Symbolic and Algebraic Computation*, 1989.

[MS94] David R. Musser and Alexander A. Stepanov. Algorithm-oriented generic libraries. *Software - Practice and Experience*, 24(7):623–642, 1994.

[MS96] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 1996.

[MSSL99] David R. Musser, Sibylle Schupp, Christoph Schwarzweller, and Rüdiger Loos. The TECTON Concept Library. Technical Report WSI99–2, Wilhelm-Schickard Institut, Universität Tübingen, 1999.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, USA, 1997.

[Muc97] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, San Francisco, California, 1997.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

[Nec97] G.C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 106–119. ACM Press, 1997.

[Nip93] Tobias Nipkow. Functional unification of higher-order patterns. In *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 64–74, Montreal, Canada, 19–23 June 1993. IEEE Computer Society Press.

[NM98] Gopalan Nadathur and Dale Miller. Higher-order logic programming. In *Handbook of Logic in AI and Logic Programming*, volume 5, pages 499–590. Oxford University Press, 1998.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, Berlin, 1999.

[NP01] Sara Negri and Jan von Plato. *Structural Proof Theory*. Cambridge University Press, 2001.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Number 2283 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2002.

[OCa03] Objective Caml 3.07. `http://caml.inria.fr`, September 2003.

[OL96] Martin Odersky and Konstantin Läufer. Putting Type Annotations to Work. In *The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 54–67. ACM, ACM Press, January 1996.

[ON99] David von Oheimb and Tobias Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*, pages 119–156. Springer, 1999. `http://isabelle.in.tum.de/Bali/papers/Springer98.html`.

[OSW99] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 1999.

[OW97] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 146–159. ACM Press, New York (NY), USA, 1997.

[Pal90] Catuscia Palamidessi. Algebraic properties of idempotent substitutions. In Mike Paterson, editor, *Automata, Languages and Programming, 17th International Colloquium, ICALP 90, Warwick University, England, July 16-20, 1990, Proceedings*, volume 443 of *Lecture Notes in Computer Science*, pages 386–399. Springer, 1990.

[Pau86] L. C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3(3):237–258, October 1986.

[Pau89] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(4):363–397, 1989.

[Pau94] Lawrence C. Paulson. *Isabelle – A Generic Theorem Prover.* Number 828 in Lecture Notes in Computer Science. Springer-Verlag, Berlin Heidelberg, 1994.

[PE88] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation (PLDI)*, volume 23 (7) of *SIGPLAN Notices*, pages 199–208, Atlanta, Georgia, June 22-24 1988. ACM Press.

[Pfe88] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 153 – 163, Snowbird, Utah, United States, July 1988. ACM Press.

[Pfe96] Frank Pfenning. The practice of logical frameworks. In Hélène Kirchner, editor, *Proceedings of the Colloquium on Trees in Algebra and Programming*, pages 119–134, Linköping, Sweden, 1996. Springer-Verlag LNCS 1059.

[Pfe01] Frank Pfenning. Logical frameworks. In Robinson and Voronkov [RV01], chapter 17.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, Cambridge, Massachusetts, 2002.

[Pot98] François Pottier. Type inference in the presence of subtyping: from theory to practice. Research Report 3483, INRIA, September 1998.

[Pot00] François Pottier. A versatile constraint-based type inference system. *Nordic Journal of Computing*, 7(4):312–347, November 2000.

[Pot01] François Pottier. Simplifying subtyping constraints: A theory. *Information and Computation*, 170(2):153–183, November 2001.

[Pra65] D. Prawitz. *Natural Deduction : A Proof-Theoretical Study.* Almqvist & Wiksell, Stockholm, 1965.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994.

[Red88] Uday S. Reddy. Objects as closures: Abstract semantics of object oriented languages. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 289–297, Snowbird, Utah, USA, July 1988. ACM Press.

[Reh97] Jakob Rehof. Minimal typings in atomic subtyping. In *Proceedings POPL'97, 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 278–291, Paris, France, January 1997. ACM.

[Rém91] Didier Rémy. Type Inference for Records in a Natural Extension of ML. research report 1431, INRIA, Rocquencourt, France, May 1991.

[Rém92] Didier Rémy. Extension ML type system with a sorted equational theory on types. Technical Report 1766, INRIA, Rocquencourt, France, October 1992.

[Rep84] T. Reps. *Generating Language-Based Environments.* M.I.T. Press, Cambridge, MA, 1984.

[Ric78] Michael M. Richter. *Logikkalküle*, volume 43 of *Studienbücher (Informatik).* B.G. Teubner, Stuttgart, 1978.

[Rit93] Dennis M. Ritchie. The development of the C language. In *The second ACM SIGPLAN conference on History of programming languages*, volume 28 (3) of *ACM Sigplan Notices*, pages 201–208. ACM Press, March 1993.

[Rob65] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[RS92] Mark Ryan and Martin Sadler. Valuation Systems and Consequence Relations. In S. Abramsky, Dov M. Gabbay, and T. S.Ẽ. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1, pages 1–78. Oxford University Press, Walton Stress, Oxford, 1992.

[RSV01] I. V. Ramakrishnan, R. C. Sekar, and Andrei Voronkov. Term indexing. In Robinson and Voronkov [RV01], chapter 26.

[RT88] Thomas W. Reps and Tim Teitelbaum. *The synthesizer generator: A system for constructing language-based editors.* Texts and Monographs in Computer Science. Springer-Verlag, New York, 1988.

[RV01] Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning.* Elsevier Science, Amsterdam, The Netherlands, 2001.

[San95] Philip S. Santas. A type system for computer algebra. *Journal of Symbolic Computation*, 1995.

[Sch96a] Sibylle Schupp. *Generic Programming Such That One can build an algebraic library.* PhD thesis, Wilhelm-Schickard Institut,Universität Tübingen, 1996.

[Sch96b] Sibylle Schupp. How to Lift a Library. Technical Report 96–7, Wilhelm-Schickard InstitutUniversität Tübingen, 1996.

[Sch97] Christoph Schwarzweller. *MIZAR verification of generic algebraic algorithms.* PhD thesis, Universität Tübingen, 1997.

[Sch02] Christoph Schwarzweller. Symbolic deduction in mathematical databases based on properties. In S. Colton and V. Sorge, editors, *Proceedings of the Second International Workshop on the Role of Automated Deduction in Mathematics (RADM2002)*, Kopenhagen, Denmark, July 2002.

[Sch03] Christoph Schwarzweller. Towards formal support for generic programming. Habilitationschrift der Fakultät für Informatik der Universität Tübingen, January 2003.

[SGM02] Clemens Szypersky, Dominik Gruntz, and Stephan Murer. *Component Software.* Component Software Series. Addison-Wesley / ACM Press, 2nd edition, 2002.

[SH87] Peter Schroeder-Heister. Structural frameworks with higher-level rules, July 1987. Habilitationsschrift, Universität Konstanz.

[Sim03] Vincent Simonet. Type inference with structural subtyping: A faithful formalization of an efficient constraint solver. In Atsushi Ohori, editor, *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS'03)*, volume 2895 of *Lecture Notes in Computer Science*, pages 283–302, Beijing, China, November 2003. Springer-Verlag.

[SL00] Jeremy Siek and Andrew Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *Proceedings First Workshop on C++ Template Programming*, Erfurt, Germany, 2000.

[SOM93] Clemens Szypersky, Stephen Omohundro, and Stephan Murer. Engineering a programming language: The type and class system of Sather. Technical report, ICSI, Berkeley, 1993.

[SS01] Peter J. Stuckey and Martin Sulzmann. A systematic approach in type system design based on constraint handling rules. Technical Report TR2001/30, The University of Melbourne, Department of Computer Science, 2001.

[SS04] P. J. Stuckey and M. Sulzmann. A unifying inference framework for Hindley/Milner with extensions. Technical Report TR12/04, The National University of Singapore, 2004.

[SSW04] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. Improving type error diagnosis. In *Proceedings of Haskell Workshop (Haskell'04)*, May 2004. To appear.

[Sto99] Frieder Stolzenburg. An algorithm for general set unification and its complexity. *Journal of Automated Reasoning*, 22(1):45–63, 1999.

[Str91] Bjarne Stroustrup. *The C++ programming language.* Addison-Wesley, Reading, Massachusetts, 2nd ed. edition, 1991.

[Str97] Bjarne Stroustrup. *The C^{++} programming language.* Addison-Wesley, Reading, Mass., 3rd edition, 1997.

[Sul00] Martin Sulzmann. *A General Framework for Hindley/Milner Type Systems with Constraints.* PhD thesis, Yale University, Department of Computer Science, May 2000.

[Sul01] Martin Sulzmann. TIE: A CHR-based type inference engine. Technical Report TR2001/27, Department of Computer Science, University of Melbourne, 2001.

[Sza69] M.E. Szabo, editor. *The collected papers of Gerhard Gentzen.* Studies in Logic and the Foundations of Mathematics. North-Holland Publishing Company, Amsterdam, 1969.

[TD01] F. Tip and T.B. Dinesh. A slicing-based approach for locating type errors. *ACM Transactions on Software Engineering and Methodology*, 10(1):5–55, January 2001.

[Thi94] Peter Thiemann. *Grundlagen der funktionalen Programmierung.* Leitfäden der Informatik. B.G. Teubner Stuttgart, 1994.

[Tof90] Mads Tofte. Type Inference for Polymorphic References. *Information and Computation*, 89:1–34, 1990.

[TR81] T. Teitelbaum and T. Reps. The Cornell Programm Synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, September 1981.

[Tro99] Anne S. Troelstra. Marginalia on sequent calculi. *Studia Logica*, 62(2), March 1999.

[TS00] A.S. Troelstra and H. Schwichtenberg. *Basic Proof Theory.* Number 43 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2nd edition, 2000.

[TTD97] J. M. Talbot, S. Tison, and P. Devienne. Set-Based Analysis for Logic Programming and Tree Automata. In *Proceedings of the Static Analysis Symposium, SAS'97*, volume 1302 of *Lecture Notes in Computer Science*, pages 127–140. Springer-Verlag, 1997.

[VR94] Peter Van Roy. 1983-1993: The wonder years of sequential Prolog implementation. *Journal of Logic Programming*, 19/20:385–441, 1994.

[Wad90] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods, Sea of Galilee, Israel*, Amsterdam, April 1990. North Holland.

[Wan87] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, 10:115–122, 1987.

[Wan91] Mitchell Wand. Type Inference for Record Concatenation and Multiple Inheritance. *Information and Computation*, 93(1):1–15, 1991.

[WB89] Philip Wadler and Stephen Blott. How to Make ad-hoc Polymorphism Less ad-hoc. In J. Hughes, editor, *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 60–76, Austin, Texas, January 1989.

[Web93] Andreas Weber. *Type systems for computer algebra.* PhD thesis, Universität Tübingen, 1993.

[Wei03] Roland Weiss. *Compiling and Distributing Generic Libraries with Heterogeneous Data and Code Representation.* PhD thesis, Wilhelm-Schickard Institut für Informatik, Eberhard-Karls Universität Tübingen, Germany., 2003.

[WF92] Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. Technical Report TR91-160, Department of Computer Science, Rice University, Houston, Texas, June 1992.

[Wic99] Philip Wickline. The Terzo interpreter for λProlog. `http://www.cse.psu.edu/~dale/lProlog/terzo`, 1999.

[Wir88] Niklaus Wirth. *Programming in Modula 2.* Texts and monographs in computer science. Springer-Verlag, Berlin ; Heidelberg, 4th edition, 1988.

[WNKN04] Martin Wildmoser, Tobias Nipkow, Gerwin Klein, and Sebastian Nanz. Prototyping proof carrying code. *Proc. 3rd IFIP Int. Conf. Theoretical Computer Science (TCS 2004)*, 2004.

[Wri94] Andrew K. Wright. *Practical Soft Typing.* PhD thesis, Rice University, Houston,Texas, August 1994.

[XP99] Hongwei Xi and Frank Pfenning. Dependent Types in Practical Programming (extended abstract). In *Symposium on the Principles of Programming Languages '99*, pages 214–227. ACM SIGACT/SIGPLAN, January 1999.