Formalisierung schematischer Algorithmen

Axel Dold
Abt. Künstliche Intelligenz
Universität Ulm
D-89069 Ulm, Germany
dold@informatik.uni-ulm.de

Zusammenfassung

Die vorliegende Arbeit beschreibt die korrekte Formalisierung eines Software-Entwicklungsschrittes mit Funktionalen höherer Ordnung am Beispiel des schematischen Algorithmus *Globalen Suche*. Wir zeigen, daß sowohl der Entwicklungsschritt selbst als auch dessen Korrektheit in ein und demselben (typtheoretischen) Formalismus dargestellt werden kann. Die Anwendungsmöglichkeit dieses Entwicklungsschrittes wird anhand eines Beispiels verdeutlicht.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Software-Entwicklung durch „schrittweise Verfeinerung“ stellt eine weitverbreitete und anerkannte Methode dar, mit formalen Methoden sichere und korrekte Software zu erhalten. Die Idee hierbei ist, ausgehend von einer formalen Anforderungsspezifikation durch schrittweise Anwendung von transformationellen korrektheitserhaltenden Entwicklungsschritten zu einem ausführbaren Programm zu gelangen, welches genau das in der Anforderungsspezifikation spezifizierte Problem löst. Diese Methodik wird insbesondere im Verbund-Projekt KORSO (Korrekte Software) [23] verfolgt.

Eine wesentliche Aufgabe hierbei stellt die Identifizierung und Formalisierung korrekter Entwicklungsschritte dar. Solche Entwicklungsschritte können sehr unterschiedliche Komplexität aufweisen. Einer oder sehr wenige mächtige komplexe Entwicklungsschritte können ausreichen, ein ausführbares Programm zu erhalten, wohingegen die Anwendung zahlreicher „elementarer“ Transformationen nötig ist, um zum selben Ergebnis zu gelangen. Diese lassen sich insbesondere zu mächtigen Schritten zusammenfassen. Die Idee der transformationellen Programmentwicklung wurde vor allem von der Münchner CIP-Gruppe [3, 4, 13] verfolgt. Sie hat Transformationstechniken zusammen mit einer Vielzahl solcher elementarer Transformationen entwickelt.

In diesem Bericht geht es um die Formalisierung und Verifizierung eines Entwicklungsschrittes mittels Funktionen höherer Ordnung in einem typtheoretischen Rahmen [8].

Die Formalisierung von Transformationen mit Hilfe von Funktionen höherer Ordnung wurde u. a. auch von Huet und Lang [6] untersucht. Dort werden Programmtransformationen für Rekursionseliminierung als Schemata zweiter Ordnung im einfach getypten λ-Kalkül ausgedrückt. Wir betrachten hier jedoch, im Gegensatz zu Huet und Lang, einen sehr „mächtigen“ generischen Entwicklungsschritt und zeigen anhand des schematischen Algorithmus *Globale Suche*, daß es in der Sprache QED möglich ist, diesen Schritt in eleganter Weise zu formalisieren und seine Korrektheit nachzuweisen.

Ein schematischer Algorithmus kann als eine allgemeine algorithmische Struktur, eine Algorithmentheorie [20], betrachtet werden, die ein globales Programmierkonzept kodifiziert, auf welcher viele Algorithmen beruhen. *Globale Suche* etwa, verallgemeinert so bekannte Suchstrategien wie Backtracking, Tiefensuche und heuristische Suche, die einen bestimmten Lösungsraum systematisch durchsuchen. Kann man ein gegebenes Problem zu einer bestimmten Form, einer sogenannten Globalen Suchtheorie, durch Hinzufügung zusätzlicher Datenstrukturen erweitern, so läßt sich sofort durch Instanziierung des schematischen Algorithmus *Globale Suche* eine Lösung des Problems angeben. Dies bedeutet also, daß die einmalige Anwendung dieses Entwicklungsschrittes ausreicht, um ein ausführbares Programm zu erhalten.

Der vorliegende Bericht basiert auf den Arbeiten von Doug Smith [17, 18, 19], der zahlreiche Algorithmentheorien untersucht und in seinem KIDS-System [19], welches eine sehr leistungsstarke Umgebung zur transformationellen Software-Entwicklung zur Verfügung stellt, implementiert hat. Teilaspekte werden dabei allerdings nur informell erfaßt. Diese Dinge werden wir im folgenden genauer formalisieren und zeigen, daß sowohl der komplette Entwicklungsschritt als auch dessen Verifizierung in rigoroser mathematischer Weise in einem typtheoretischen Formalismus dargestellt werden kann.

Christoph Kreitz hat in seiner Habilitationsschrift [7] neben zahlreichen elementaren Theorien insbesondere Programmierkonzepte, Entwicklungsschritte und bestehende Ansätze zur Programmsynthese in NUPRL [2] formalisiert, u. a. auch den in diesem Bericht betrachteten schematischen Algorithmus *Globale Suche*. Die vorliegende Arbeit ist jedoch in einem anderen Kontext und unabhängig davon entstanden.

Dieser Bericht ist wie folgt gegliedert:
Im zweiten Kapitel werde ich kurz auf die verwendete Spezifikationssprache QED eingehen, welche eine Erweiterung des (typtheoretischen) Kalküles ECC [8] darstellt. Es werden die wesentlichen Konzepte dieser Sprache erläutert, soweit sie für die dann folgende Formalisierung erforderlich sind. Im dritten Kapitel geben wir an, in welcher Form ein konkretes Problem zu spezifizieren ist. Ein Problem wird dabei im wesentlichen durch dessen Ein-/Ausgabeverhalten deskriptiv beschrieben. Im vierten Kapitel definieren wir eine Erweiterung des Problems um Datenstrukturen, die zur Durchführung der globalen Suche notwendig sind und fassen diese zu einer Spezifikation *global_search_theory* zusammen. Im fünften Kapitel formalisieren wir den schematischen Algorithmus, der im wesentlichen eine mit einer *global_search_theory* parametrisierte Funktion darstellt, die genau das Suchverfahren durchführt. Die Korrektheit und Terminierung des schematischen Algorithmus wird im sechsten Kapitel mit Hilfe des Spezifikations- und Beweissystems PVS [11] durchgeführt. Dabei wurde die QED Spezifikation nach PVS übertragen, und die sich ergebenen Beweisverpflichtungen mit Hilfe des PVS Gentzen-Beweisers, welcher adäquate Beweisunterstützung für Logik höherer Ordnung bietet, bewiesen. Die Anwendungsmöglichkeit der globalen Suche wird anhand eines einfachen Beispiels im siebten Kapitel demonstriert. Insbesondere wird gezeigt, welche Beweisverpflichtungen sich durch die Anwendung des Entwicklungsschrittes ergeben. Abschließend werden die Resultate kurz zusammengefaßt und ein Ausblick gegeben.

Danken möchte ich dem Leiter unserer KORSO-Gruppe Herrn von Henke und meinen Kollegen (Malte Grosse, Detlef Schwier, Martin Strecker und Harald Rueß) für deren kritische und hilfreiche Anregungen und Bemerkungen zum vorliegenden Bericht. Diese Arbeit wurde vom BMFT im Rahmen des KORSO-Verbundprojektes unterstützt.

2 Elemente der Sprache QED

In diesem Abschnitt beschreiben wir kurz die verwendete Sprache QED. Detaillierte Angaben und Erläuterungen findet man in der Sprachbeschreibung [14] und dem Semantikpapier [16]. Die Abb. 1 faßt die wesentlichen Sprachelemente zusammen. QED ist eine auf einer Typtheorie [8, 10] basierende Spezifikationssprache, die einen Rahmen zur modularen formalen Programmentwicklung, Verifikation und funktionalen Programmierung zur Verfügung stellt. Aufgrund des ausdrucksmächtigen zugrundeliegenden Typkonzeptes können mathematische Sachverhalte in natürlicher und eleganter Weise formalisiert werden. Einige wichtige Sprachkonzepte stammen aus dem Spezifikations- und Verifikationssystem PVS [11]. Das Typsystem umfaßt vordefinierte Standarddatentypen für Boolesche Ausdrücke ($\mathbb{B}$), natürliche Zahlen ($\mathbb{N}$), polymorphe Listen und Mengen zusammen mit elementaren Operationen. Weiterhin gibt es Typkonstruktoren, mit denen sich so fundamentale Konzepte wie abhängige Funktions- und Recordtypen, semantische Subtypen, abstrakte Datentypen, Spezifikationen und induktive Datentypen definieren lassen. Über das Prinzip des *propositions-as-types* [5] läßt sich eine (intuitionistische) Logik höherer Ordnung in das Typsystem einbetten. Logische Formeln werden also als die Typen ihrer Beweise betrachtet. Die Sprache stellt weiterhin Konzepte zur Verfügung, wie sie in gängigen funktionalen Programmiersprachen vorhanden sind. Solche Konzepte umfassen *let*-Konstrukte, Bedingungen, allgemeine Rekursion und primitive Rekursion höherer Ordnung. Alle Objekte müssen bezüglich eines Kontextes Γ wohlgetypt sein, d.h. jedes Objekt a hat im Kontext Γ einen eindeutigen (prinzipalen) Typ A ($\Gamma \vdash a : A$).

QED

Kontext Γ : Liste von Deklarationen $a : A$ und Definitionen $f := t$

Universen: *Prop* für logische Formeln, *Type* für Datentypen

Logik: $\wedge, \vee, \Rightarrow, \Leftrightarrow, \neg, \forall, \exists, =_{Leibniz}$

Typen, kanonische Objekte

- **Standarddatentypen:** $\mathbb{B}, \mathbb{N}$, polymorphe Listen
- **Recordtyp:** $\ll a_1 : A_1, \ldots, a_n : A_n \gg$
  - Tupel: $(a_1, a_2, \ldots, a_n)$
- **Kartesisches Produkt:** $A \times B$
  - (a, b) mit $a : A$ und $b : B$
- **Funktionstyp:** $[a : A] \Rightarrow B$ und $A \to B$
  - Funktionen: $\lambda\ a : A.\ M$ mit $\Gamma, a : A \vdash M : B$
- **Semantische Subtypen:** $\{a : A \mid P(a)\}$
  - (a, p) mit $a : A$ und $p : P(a)$
- **Spezifikationen, Strukturen:**
  - SPEC $x_1 : A_1, \ldots, x_n : A_n$ WITH $ax_1, ax_2, \ldots, ax_m$ END
  - STRUC $x_1 := a_1, \ldots, x_n := a_n\ (p_1, \ldots, p_m)$ END
  - mit $p_i : ax_i[x_1 \leftarrow a_1, \ldots, x_n \leftarrow a_n]$
- **Wohlfundierte Datentypen:** DATATYPE $X : T.\ M$

Applikation: $f(t)$

Let: LET $f := t$ IN M END

Rekursive Funktionen: FIX $f : A.\ M$ und primitive Rekursion.

Abbildung 1: Die Spezifikationssprache *QED*

3 Allgemeine Problembeschreibung

In diesem Abschnitt geben wir an, in welcher Form die Anforderungsspezifikation eines Problems erfolgt. Wir werden dabei im folgenden auf die vordefinierten Datentypen der Sprache QED zurückgreifen. Weiterhin setzen wir voraus, daß der Datentyp der parametrisierten (endlichen) Mengen mit den folgenden zugehörigen Operationen gegeben ist:

- $Set(T)$ bezeichnet (den Typ der) Mengen mit Elementen vom Typ T
- $\varnothing_T$ bezeichnet die leere Menge (über T).
- $insert(elem, s)$ fügt das Element $elem$ in die Menge s ein.
- $\in$ ist der Test auf Enthaltensein
- $card(s)$ Kardinalitätsfunktion
- $=_{Set}$ bezeichnet die (entscheidbare) Gleichheit auf Mengen[1]

Die Anforderungsspezifikation eines Problems beschreiben wir anhand dessen Ein-/Ausgabeverhaltens. Hierfür verwenden wir einen (abhängigen) Record-Typ mit vier Feldern der folgenden Form

$$Problemspec := \ll D : Type, R : Type, I : D \to Prop, O : D \times R \to Prop \gg$$

Hierbei bezeichnen

- D den Domaintyp,
- R den Ergebnistyp,
- $I(x)$ die Eingabebedingung, die die Eingabe x auf diejenigen Objekte des Domains beschränkt, für welche diese Bedingung gilt.
- $O(x, z)$ die Ausgabebedingung, welche genau dann wahr ist, wenn $z : R$ eine Lösung in Abhängigkeit von $x : D$ ist.

Eine Realisierung einer *Problemspec* ist dann ein als 4-Tupel $(d_{impl}, r_{impl}, i_{impl}, o_{impl})$ beschriebenes Problem, d.h. für D und R werden konkrete Typen und für I und O konkrete Prädikate angegeben. Gesucht sind nun alle diejenigen Objekte z von R, für welche die Bedingung $O(x, z)$ erfüllt ist. Das Problem, eine einzelne Lösung zu finden (etwa eine optimale Lösung) ist dann ein Spezialfall, wobei die Lösungsmenge genau ein Element enthält. Wir formalisieren dies als eine Funktion *req_spec*, die eine Problembeschreibung *Problemspec* als Eingabe erhält und eine Proposition liefert, gerade die Aussage, daß für jedes Objekt des Domains, welches die Eingabebedingung erfüllt, eine Menge S existiert, so daß für jedes Element dieser Menge die Input/Output Relation erfüllt ist.

$$\begin{aligned} req\_spec := \ & \lambda\ (D, R, I, O) : Problemspec. \\ & \forall\, x : D.\ I(x) \Rightarrow \exists\, S : Set(R).\ \forall\, elem : R. \\ & ((elem \in S) = true) \Leftrightarrow O(x, elem) \end{aligned}$$

[1] Hiebei setzen wir natürlich voraus, daß ebenso eine entscheidbare Gleichheit auf dem Elementtyp T gegeben ist.

Die Anforderungsspezifikation eines Problems ist also durch solch ein Viertupel eindeutig spezifiziert. Ein (konstruktiver) Beweis von $req\_spec\,(d_{impl}, r_{impl}, i_{impl}, o_{impl})$ liefert eine Funktion f, die zu jedem x aus d_{impl}, welches die Eingabebedingung i_{impl} erfüllt, die gesuchte Menge S berechnet. Ein solcher Beweis ist im allg. sehr schwierig, wenn nicht sogar unmöglich zu finden. Doch wir werden später sehen, daß durch Erweiterung der Anforderungsspezifikation um spezielle Datenstrukturen und Operationen, eine Lösung durch Instanziierung eines schematischen Algorithmus synthetisiert werden kann. Dieser Algorithmus basiert auf diesen zusätzlichen Datenstrukturen.

Im nächsten Abschnitt geben wir eine solche Theorie als Erweiterung einer Problembeschreibung an, auf welcher sich dann ein schematischer Algorithmus definieren läßt.

4 Globale Suchtheorie

In diesem Abschnitt betrachten wir eine spezielle Algorithmentheorie, die *Globale Suche*. Eine Algorithmentheorie kann als eine Zusammenfassung von Algorithmen betrachtet werden, die eine gewisse Ähnlichkeitsstruktur aufweisen. Für diese Klassen lassen sich dann gemeinsame Realisierungen finden [20]. Man kann eine solche Theorie als Erweiterung eines konkreten Problems um zusätzliche Datenstrukturen betrachten, die zur Lösung des Problems erforderlich sind. Für diese Klassen lassen sich allgemeine Lösungsprinzipien, die im nächsten Abschnitt betrachteten schematischen Algorithmen, ableiten. Kann man nun das konkrete Problem in solch ein Schema einbetten (durch entsprechende Realisierungen der Datentypen und Funktionen) so können die allgemeinen Lösungsprinzipien durch Instanziierung mit dem konkreten Problem verwendet werden, und man erhält sofort eine Lösung.

Globale Suche stellt eine Generalisierung so bekannter Suchstrategien wie *Backtracking*, *binäre Suche*, *Branch-and-Bound* und heuristische Suche dar [18]. Sie beruht auf der Idee der Aufspaltung von Mengen in Untermengen und der Extrahierung von Lösungen aus diesen Mengen. Das Verfahren läßt sich als eine Suche in einem Suchbaum repräsentieren, wobei Knoten den Mengen und die Söhne dieser Knoten den Untermengen entsprechen. Die Nachfolgerrelation bezeichnen wir im folgenden in Analogie zu Smith [18] mit *split*?[2]

Die oben erwähnten Mengen, auf welchen das Suchverfahren arbeitet, können im allg. nicht explizit repräsentiert werden, sie werden daher implizit in Form von *Mengendeskriptoren* dargestelllt. Den Typ dieser Mengendeskriptoren bezeichnen wir mit S. Die Suche arbeitet also auf diesem Typ. Um den Typ S auf bestimmte Werte einzuschränken, die mit der Eingabe vom Typ D konform sind, benötigen wir ein Prädikat J. J ist genau dann wahr, falls ein Deskriptor aus S in Abhängigkeit von der Eingabe x gültig ist.

Wir wollen eine Suche im Suchraum R durchführen, verwenden hierfür aber einen anderen Typ, den Typ der Mengendeskriptoren. Deshalb benötigen wir eine Art Schnittstelle zwischen diesen Typen. Das Prädikat *satisfies* realisiert eine solche Schnittstelle. Es ist genau dann wahr, falls ein Objekt des Zielraumes R sich in der durch den gegebenen Mengendeskriptor beschriebenen Menge befindet. Diese Objekte von R sind also mögliche Lösungen des Problems (D, R, I, O).

Das Prädikat *extract*? beschreibt, welche Elemente aus einer gegebenen Menge direkt extrahiert werden können. Die tatsächlichen Lösungen sind dann alle aus der Menge extrahierten

[2] Wir brauchen hier auch eine Funktion *split*, die uns explizit aus einer gegebenen Menge diese Untermengen berechnet.

Elemente, welche die Problembedingung O erfüllen. Hierfür führen wir die Funktion *extract* ein.[3]

Um den Wurzelknoten des Suchbaumes, d. h. die Initialmenge zu generieren, die dann den Ausgangspunkt der globalen Suche darstellt und implizit alle Lösungen enthält, verwenden wir die Funktion *init*, welche uns in Abhängigkeit der Eingabe den Initialdeskriptor berechnet.

Die so eingeführten und für das Suchverfahren benötigten Datentypen und Funktionen fassen wir in einer Spezifikation *global_search_theory* zusammen. Die vollständige Spezifikation wird in Abb. 2 dargestellt. Im Prinzip handelt es sich hierbei um eine mit einer Problembeschreibung parametrisierte Spezifikation, welche wir in Form einer Funktion, die ein *Problemspec* als Eingabe erhält und eine Spezifikation liefert, ausdrücken. Die Axiome schränken dabei die möglichen Realisierungen auf die für die Suche notwendigen und sinnvollen ein, insbesondere sind wir daran interessiert eine korrekte und vollständige Lösungsmenge zu berechnen. Nachfolgend seien die Axiome der Spezifikation kurz erläutert:

ax1: Der Initialdeskriptor ist ein gültiger Deskriptor.

ax2: Falls s ein gültiger Deskriptor ist, dann sind auch alle seine Nachfolger gültige Deskriptoren.

ax3: Alle Lösungen sind in der durch den Initialdeskriptor beschriebenen Menge enthalten.

ax4: Ein Element des Ergebnistyps in genau dann in der durch s beschriebenen Menge enthalten, wenn es aus einem Nachfolger von s extrahiert werden kann.[4]

ax5: Das Ergebnis des Aufrufs von *extract* ist eine Menge, so daß jedes Element dieser Menge aus dem Knoten extrahiert werden kann und die Problembedingung $O(x, z)$ erfüllt ist.

ax6: *split* berechnet für einen gegebenen Knoten seine Nachfolger (Untermengen).

ax7: Es gibt keine Zyklen bezüglich der Nachfolgerrelation.

ax8: Jeder Deskriptor hat höchstens einen Vorgänger.

Wir müssen noch das Prädikat *transcl* definieren, welches eine Relation als Eingabe erhält und genau die transitive Hülle dieser Relation realisiert. Das Prädikat verwendet ein Hilfsprädikat *hered*, welches eine Eigenschaft P in Abhängigkeit der gegebenen Relation vererbt, d.h. falls x die Eigenschaft P besitzt und $R(x, y)$ gilt, so soll y ebenfalls die Eigenschaft P besitzen. Die Formalisierung dieser Prädikate zeigt die Abb. 3. Die Korrektheit dieser Formalisierung der transitiven Hülle ergibt sich durch den Beweis des *soundness* Theorems: die transitive Hülle einer Relation ist transitiv und die Minimalität durch den Beweis des *minimality* Theorems: *transcl*(*Rel*) ist die kleinste transitive Erweiterung von *Rel*.

Die Hauptschwierigkeit, ein gegebenes Problem zu einer globale Suchtheorie zu erweitern, besteht in der Definition des Repräsentationstyps S, also dem Typ der Knoten im Suchbaum, zusammen mit den Prädikaten *split*? und *extract*?. Die restlichen Elemente der Spezifikation lassen sich dann leicht angeben, insbesondere können die Funktionen *split* und *extract* im allg. sehr einfach aus den Prädikaten *split*? bzw. *extract*? bestimmt werden. Wir werden dies am Beispiel im 7. Kapitel noch genauer sehen.

[3]Smith verwendet nur Mengenbeschreibungen. Wir hingegen benötigen eine explizite Operation, welche die Lösungen durch Extrahierung und Testen herausfiltert.

[4]Das so erhaltene Element muß dann noch auf Gültigkeit überprüft werden.

$global\_search\_theory := \ \lambda\ (D, R, I, O) : Problemspec.$

SPEC

$S : Type$

$J : D \times S \to Prop$

$init : D \to S$

$satisfies : R \times S \to Prop$

$split? : D \to (S \times S \to Prop)$

$split : D \times S \to Set(S)$

$extract? : R \times S \to Prop$

$extract : D \times S \to Set(R)$

WITH

$ax1 : \forall\, x : D.\ I(x) \Rightarrow J(x, init(x)),$

$ax2 : \forall\, x : D, r, s : S.$
$(I(x) \wedge J(x, r) \wedge split?(x)(r, s)) \Rightarrow J(x, s),$

$ax3 : \forall\, x : D, z : R.$
$(I(x) \wedge O(x, z)) \Rightarrow satisfies(z, init(x)),$

$ax4 : \forall\, x : D, r : S, z : R.$
$(I(x) \wedge J(x, r)) \Rightarrow$
$satisfies(z, r) \Leftrightarrow (extract?(z, r) \vee$
$\exists\, s : S.\ transcl(split?(x))(r, s) \wedge extract?(z, s)),$

$ax5 : \forall\, x : D, r : S.$
$(I(x) \wedge J(x, r)) \Rightarrow$
$\forall\, z : R.\ (z \in extract(x, r)) = true \Leftrightarrow extract?(z, r) \wedge O(x, z),$

$ax6 : \forall\, x : D, r : S.$
$(I(x) \wedge J(x, r)) \Rightarrow$
$\forall\, s : S.\ (s \in split(x, r)) = true \Leftrightarrow split?(x)(r, s)$

$ax7 : \forall\, x : D, s : S.$
$(I(x) \wedge J(x, s)) \Rightarrow \neg\ transcl(split?(x))(s, s),$

$ax8 : \forall\, x : D, r_1, r_2, s : S.$
$(I(x) \wedge J(x, r1) \wedge J(x, r2)) \Rightarrow$
$(split?(x)(r_1, s) \wedge split?(x)(r_2, s)) \Rightarrow r_1 = r_2$

END

Abbildung 2: Die *Globale Suchtheorie*

$$
\begin{aligned}
&\mathit{trans?} := \ \lambda\ T \mid \mathit{Type}, \mathit{Rel} : T \times T \to \mathit{Prop}.\\
&\quad \forall\, x, y, z : T.\ \mathit{Rel}(x, y) \Rightarrow \mathit{Rel}(y, z) \Rightarrow \mathit{Rel}(x, z)\\
&\mathit{hered} := \ \lambda\ T \mid \mathit{Type},\ \mathit{Rel} : T \times T \to \mathit{Prop},\ P : T \to \mathit{Prop}.\\
&\quad \forall\, x, y : T.\ P(x) \Rightarrow \mathit{Rel}(x, y) \Rightarrow P(y)\\
&\mathit{transcl} := \ \lambda\ T \mid \mathit{Type},\ \mathit{Rel} : T \times T \to \mathit{Prop},\ (x, y) : T \times T.\\
&\quad \forall\, P : T \to \mathit{Prop}.\ \mathit{hered}(\mathit{Rel}, P) \Rightarrow\\
&\quad (\forall\, u : T.\ \mathit{Rel}(x, u) \Rightarrow P(u)) \Rightarrow P(y)\\
&\mathit{soundness} := \forall\ T \mid \mathit{Type},\ \mathit{Rel} : T \times T \to \mathit{Prop}.\ \mathit{trans?}(\mathit{transcl}(\mathit{Rel}))\\
&\mathit{minimality} := \forall\ T \mid \mathit{Type},\ \mathit{Rel} : T \times T \to \mathit{Prop},\\
&\qquad\qquad \mathit{Rel}_{tr} : \{T \times T \to \mathit{Prop} \mid \mathit{trans?}(\mathit{Rel}_{tr})\}.\\
&\qquad\qquad (\forall\, x, y : T.\ \mathit{Rel}(x, y) \Rightarrow \mathit{Rel}_{tr}(x, y)) \Rightarrow\\
&\qquad\qquad \forall\, x, y : T.\ \mathit{transcl}(\mathit{Rel})(x, y) \Rightarrow \mathit{Rel}_{tr}(x, y)
\end{aligned}
$$

Abbildung 3: Spezifikation der transitiven Hülle

Hat man konkrete Prädikate und Funktionen $\mathit{satisfies}_{impl}, \mathit{split}_{impl}$ usw. als ein STRUC - Konstrukt vom Typ $\mathit{global\_search\_theory}(d_{impl}, r_{impl}, i_{impl}, o_{impl})$ angegeben, so werden Beweisverpflichtungen generiert. Dies sind im wesentlichen genau die mit den konkreten Funktionen instanziierten Axiome der Spezifikation. Die im nächsten Abschnitt angegebene schematische Funktion F_{gs} implementiert dieses Verfahren mit der oben definierten Datenstruktur.

5 Schematischer Algorithmus

Der schematische Algorithmus, welcher auf den im letzten Abschnitt definierten Datenstrukturen basiert und das Verfahren der globalen Suche implementiert, arbeitet in folgender Weise:

Aus einer Menge von aktiven Knoten wird einer ausgewählt. Es werden aus diesem Knoten, der ja eine bestimmte Menge repräsentiert, alle extrahierbaren Lösungen berechnet und zur Lösungsmenge hinzugefügt. Die Nachfolger (Untermengen) des betrachteten Knotens werden zur aktiven Menge hinzugefügt und der betrachtete Knoten daraus entfernt. Das Verfahren wiederholt sich, solange es noch nicht betrachtete Knoten in der aktiven Menge gibt:

1. Die aktive Menge besteht nur aus dem Knoten *init(x)*.
2. Die Initial-Lösungsmenge ist anfangs leer.
3. Solange die aktive Menge nicht leer ist:
   (a) Wähle einen Knoten r aus der aktiven Menge aus.
   (b) Eliminiere diesen Knoten und füge dessen Nachfolger hinzu.
   (c) Falls Lösungen aus dem aktuellen Knoten r extrahiert werden können, füge diese zur Lösungsmenge hinzu.
4. Das Ergebnis des Verfahrens ist die berechnete Lösungsmenge.

Wir realisieren einen solchen Algorithmus als eine Funktion F_{gs}, die im wesentlichen eine globale Suchtheorie als Eingabe erhält und einen Algorithmus (Funktion) liefert, der für die konkreten Werte das Verfahren der globalen Suche durchführt, siehe Abb. 4. Um die Terminierung

$$
\begin{array}{l}
F_{gs} := \ \lambda\ (D, R, I, O) : Problemspec, \\
\quad gs : global\_search\_theory(D, R, I, O), \\
\quad \text{LET} \\
\quad F\_Type := \ll aktiv : Set(gs.S), \\
\qquad\qquad\quad solution : Set(R), \\
\qquad\qquad\quad x : \{D \mid Invariant((D, R, I, O), gs, aktiv, solution, x)\} \gg \\
\quad \text{IN} \\
\lambda\ arbsplit : [m : \{Set(gs.\ S) \mid empty?(m) = false\}] \Rightarrow \\
\qquad \ll elem : \{gs.\ S \mid (elem \in m) = true\}, \\
\qquad\quad rest : \{Set(gs.\ S) \mid rest =_{Set} (m - elem)\} \gg, \\
\quad huelle : [(aktiv, solution, x) : F\_Type] \Rightarrow \\
\qquad \{M : Set(gs.\ S) \mid \\
\qquad\quad \forall\, t : gs.\ S.\ (t \in M) = true \Leftrightarrow \\
\qquad\quad (t \in aktiv) = true \vee \\
\qquad\quad \exists\, r : \{gs.\ S \mid (r \in aktiv) = true\}. \\
\qquad\quad transcl(split?(x))(r, t)\}. \\
\quad \text{FIX}\ f : F\_Type \to Set(R).\ \ \lambda\ (aktiv, solution, x) : F\_Type. \\
\quad \text{IF}\ (empty?(aktiv))\ \text{THEN}\ solution \\
\qquad \text{ELSE} \\
\qquad \text{LET} \\
\qquad\quad (r, A_1) := arbsplit(aktiv), \\
\qquad\quad Newaktiv := A1 \cup gs.split(x, r), \\
\qquad\quad Newsolution := solution \cup gs.extract(x, r) \\
\qquad \text{IN} \\
\qquad\quad f(Newaktiv, Newsolution, x) \\
\qquad \text{END} \\
\quad \text{MEASURE} \\
\qquad \lambda\ (aktiv, solution, x) : F\_Type.\ card(huelle(aktiv, solution, x)) \\
\quad \text{END}
\end{array}
$$

Abbildung 4: Der schematische Algorithmus *Globale Suche*

$$
\begin{aligned}
\mathit{Invariant} := \ & \lambda\ (D, R, I, O) : \mathit{Problemspec}, \\
& \quad \mathit{gs} : \mathit{global\_search\_theory}(D, R, I, O), \\
& \quad \mathit{aktiv} : \mathit{Set}(\mathit{gs}.\ S), \\
& \quad \mathit{solution} : \mathit{Set}(R), \\
& \quad x : D. \\
& I(x) \wedge \forall\, s : \mathit{gs}.\ S.\ (s \in \mathit{aktiv} = \mathit{true}) \Rightarrow \mathit{gs}.\ J(x, s) \wedge \\
& \forall\, z : R.\ (z \in \mathit{solution} = \mathit{true}) \Rightarrow O(x, z) \wedge \\
& \forall\, s_1, s_2 : \{s : \mathit{gs}.\ S \mid s \in \mathit{aktiv} = \mathit{true}\}.\ \neg\ \mathit{trancl}(\mathit{gs.split?}(x))(s_1, s_2)
\end{aligned}
$$

Abbildung 5: Invariante der Funktion F_{gs}

des Verfahrens sicherzustellen, fügen wir als Parameter noch eine Funktion *huelle* ein, die zu der aktiven Menge jeweils deren (endliche) Menge M von Knoten in der (reflexiven) transitiven Hülle bestimmt. Damit erhalten wir eine endliche Tiefe des Suchbaumes. Eine endliche Breite bekommen wir implizit durch die Funktion *split*, welche zu jedem Knoten dessen endliche (direkte) Nachfolgermenge berechnet. Hier zeigt sich die elegante Formulierung mit Hilfe des Konzeptes des semantischen Subtyps, um eine solche Funktion genau zu spezifizieren.

Die rekursive Funktion spezifizieren wir mit dem *FIX*-Operator. Um die Wohlgetyptheit zu garantieren, wird außerdem eine Maßfunktion benötigt, welche die Terminierungsordnung definiert. Als Terminierungsfunktion (Maßfunktion) wählen wir die Kardinalität der (reflexiven) transitiven Hülle aller Knoten aus der aktiven Menge. Da jedesmal der betrachtete Knoten aus der aktiven Menge entfernt wird, verringert sich diese Kardinalität bei jedem (rekursiven) Funktionsaufruf genau um 1.

Die Funktion *arbsplit*, welche wir ebenfalls als Parameter zu F_{gs} hinzunehmen, wählt einen Knoten r aus der aktiven Menge aus und gibt ein Tupel bestehend aus r und der Restmenge A_1 zurück. Die Art der Auswahl dieses Elementes implementiert die verschiedenen Suchstrategien wie Tiefen- oder Breitensuche. Tiefensuche könnte man etwa so realisieren, daß *arbsplit* immer dasjenige Element aus *aktiv* wählt, welches als letztes durch *split* hinzugefügt wurde.

Weiterhin müssen bei jedem Funktionsaufruf bestimmte Bedingungen erfüllt sein, nämlich daß

- jeder Knoten der aktiven Menge ein gültiger Mengendeskriptor ist und
- jedes Element aus der Lösungsmenge tatsächlich eine Lösung darstellt, d.h. für dieses Element gilt die Ein-/Ausgabebedingung O und
- für je zwei beliebige Knoten t_1, t_2 aus der aktiven Menge gilt, daß t_2 nicht in der transitiven Hülle von t_1 liegt.

Diese Bedingungen fassen wir als *Invariante* zusammen und definieren hiermit einen den Domain D einschränkenden semantischen Untertyp. Abb. 4 zeigt die Formalisierung von F_{gs}. Wir geben die Invariante in Abb. 5 an.

Um die Korrektheit des Entwicklungsschrittes *Globale Suche* sicherzustellen, müssen wir zeigen, daß für beliebige Problemspezifikationen und Globale Suchtheorien, die instanziierte Funktion F_{gs} das Problem löst, d.h. *F_inst* berechnet genau die Menge aller $z : R$, für die die Bedingung O mit Input x erfüllt ist. Die Initialwerte des Verfahrens sind dabei neben dem Parameter x für

$$
\begin{array}{l}
soundness\_theorem := \\
\quad \forall\,(D, R, I, O) : Problemspec, \\
\quad\; gs : global\_search\_theory(D, R, I, O), \\
\quad\; arbsplit : \ldots,\ huelle : \ldots \\
\quad\; x : \{D \mid I(x)\},\ y : R. \\
\quad\; \text{LET} \\
\qquad\; F\_inst := F_{gs}((D, R, I, O), gs, arbsplit, huelle), \\
\qquad\; init\_set := insert(gs.\ init(x), \varnothing_S), \\
\qquad\; init\_sol := \varnothing_R, \\
\qquad\; sol\_set := F\_inst(init\_set, init\_sol, x) \\
\quad\; \text{IN} \\
\qquad\; ((y \in sol\_set) = true) \Leftrightarrow O(x, y) \\
\quad\; \text{END}
\end{array}
$$

Abbildung 6: Korrektheit der *Globalen Suche*

- *aktiv*: die Menge, welche nur aus dem Initialdeskriptor *init(x)* besteht
- *solution* : $\varnothing_R$

Dieses Theorem läßt sich im Formalismus QED in einfacher Weise explizit formalisieren wie Abbildung 6 zeigt.[5]

6 Verifikation der globalen Suche

Um die im letzten Abschnitt angegebene Korrektheitsaussage formal zu verifizieren, ist eine adäquate Beweisunterstützung unabdingbar. Da diese für die Sprache QED momentan nicht zur Verfügung steht, wurde für die Durchführung des Beweises das Spezifikations- und Beweissystem PVS [11] ausgewählt, welches ebenfalls eine Logik höherer Ordnung mit einem mächtigen ausdrucksstarken Typsystem und ähnliche Konzepte wie QED (z. B. subtyping, abhängige Funktionen, Tupel- und Recordtypen) zur Verfügung stellt. Desweiteren beinhaltet PVS einen leistungsstarken (interaktiven) Gentzen-Beweiser für Logik höherer Ordnung, welcher mächtige Entscheidungsprozeduren für Arithmetik und Standarddatentypen enthält, insbesondere läßt sich der Automatisierungsgrad durch Verwendung von Taktiken erheblich erhöhen. PVS enthält Konstrukte zur modularen Organisation und Zusammenfassung von mathematischen oder programmiersprachlichen Konstrukten in Theorien. Die Behandlung ist allerdings in der derzeitigen Version rein syntaktischer Natur, und im Unterschied zu QED sind diese Theorien nicht in die zugrundeliegende Logik integriert. Parametrisierte Spezifikationen sind daher eher als variable Kontexte zu sehen, die, instanziiert mit aktuellen Parametern, erst eine korrekte Datenbasis darstellen. Ebenso ist es nicht möglich, Theorien als Parameter zu übergeben. Allerdings lassen sich die formalen Parameter durch Verwendung von sogenannten "assumptions" einschränken. Hierbei handelt es sich um Formeln, die für jede Instanz der Theorie erfüllt sein müssen.

[5] Die Typdefinitionen von *arbsplit* und *huelle* wurden hier aus Lesbarkeitsgründen weggelassen.

6.1 Umsetzung einer QED-Spezifikation in PVS

Im folgenden beschreiben wir stichpunktartig die Umsetzung einer QED-Formalisierung in eine entsprechende PVS-Formalisierung. Die folgende Aufzählung orientiert sich an Abb. 1.

1. Ein Kontext Γ bestehend aus Deklarationen und Definitionen wird in eine oder mehrere PVS-Theorien übertragen, die dann 'importiert' werden können. Die Importierung entspricht im wesentlichen der Hinzufügung von Definitionen und Deklarationen zur aktuellen Theorie.

2. Für Datentypen stellt PVS den Typ TYPE zur Verfügung. Formeln erhalten allesamt den Typ BOOL. PVS beinhaltet eine (klassische) Logik höherer Ordnung und stellt alle üblichen Konnektoren zur Verfügung.

3. Die Standarddatentypen (*nat*, *set*$[T]$, *list*$[T]$, *bool* usw.) zusammen mit entsprechenden Operationen sind in PVS vordefiniert.

4. Recordtypen der Form $\ll a_1 : A_1, \ldots, a_n : A_n \gg$ werden als $[a_1 : A_1, \ldots a_n : A_n]$ notiert. Zu jedem Record-Typ gibt es entsprechende Projektionsfunktionen *proj_i*.

5. Das Kartesische Produkt $A \times B$ wird als $[A, B]$ notiert.

6. Funktionstypen haben die Form $[a_1 : A_1, \ldots, a_n : A_n \to B]$.

7. Die Notation der semantischen Subtypen entspricht derjenigen in QED.

8. QED-Spezifikationen

   $$name := \text{SPEC } x_1 : A_1, \ldots, x_n : A_n \text{ WITH } ax_1, ax_2, \ldots ax_m \text{ END}$$

   würde man in PVS etwa in der Art

```
name [x1 : A1,
      x2 : A2,
      ...
      xn : An] : THEORY
BEGIN
 ASSUMING
  ax1 : ASSUMPTION ...
  ax2 : ASSUMPTION ...
  ...
  axm : ASSUMPTION ...
 ENDASSUMING
END name
```

   umsetzen. Eine Realisierung einer solchen (parametrisierten) Theorie ist eine Instanz, welche die Axiome erfüllt. Durch Importierung der instanziierten Theorie werden eine Reihe von Beweisverpflichtungen generiert. Dies sind gerade die obigen Axiome instanziiert mit den spezifischen Werten.

6.2 Umsetzung der Globalen Suchtheorie

Wir formalisieren die Globale Suche als eine mit einer Problembeschreibung D, R, I, O und den benötigten Datentypen und Funktionen $(S, J, satisfies, ...)$ parametrisierten Theorie und fassen alle Datenstrukturen, insbesondere den schematischen Algorithmus F_{gs} in dieser Theorie zusammen. Weiterhin haben wir die folgenden Änderungen durchgeführt:

1. Die Theorie der transitiven Hülle einer Relation wird importiert. Die PVS-Formalisierung ist im Anhang B angegeben.
2. Die Axiome $ax1$ bis $ax8$ werden im wesentlichen eins zu eins in Form von 'assumptions' übertragen.
3. Die Funktion *huelle*, die in der QED-Spezifikation durch einen semantischen Subtyp spezifiziert wird, wurde mittels eines Axioms *huelle_ax* beschrieben.
4. Die Funktion *arbsplit* wird durch die beiden Axiome *arbsplit*1 und *arbsplit*2 beschrieben. Dabei spezifizieren wir *arbsplit* nur für den Fall, daß die Funktion auf eine nichtleere Menge angewandt wird, was bei der Benutzung von *arbsplit* auch immer der Fall ist.
5. Für die Maßfunktion benötigen wir die Kardinalität von (endlichen) Mengen. Hier brauchen wir lediglich die Tatsache, daß die Kardinalität einer echten Teilmenge einer Menge kleiner ist als die Kardinalität dieser Menge.
6. Die schematische Funktion F_{gs} läßt sich nahezu identisch übertragen.
7. Die Beweise, die in der klassischen Logik höherer Ordnung von PVS durchgeführt werden, lassen sich in die QED-Logik übertragen. Bei den verwendeten Prädikaten handelt es sich allesamt um entscheidbare Prädikate,[6] Fallunterscheidungen wurden nur auf entscheidbaren Prädikaten durchgeführt und alle (Existenz-)Beweise konstruktiv geführt.

Die PVS-Theorie mit allen benötigten Datenstrukturen, Lemmata und Korollaren zur Globalen Suche ist im Anhang A angegeben.

6.3 Durchführung des Beweises

Im folgenden beschreiben wir alle für den Beweis des Korrektheitstheorems benötigten Beweisverpflichtungen, insbesondere die Typkorrektheitsbedingungen, welche in PVS automatisch generiert werden. Das Haupttheorem *korrekt*, welches die Korrektheit des schematischen Algorithmus formalisiert und oben in QED angegeben ist, wird im wesentlichen mit Hilfe der Lemmata *lemma*1 und *lemma*11 bewiesen. *lemma*1 und *lemma*11 beschreiben, welches Ergebnis F_{gs} für allgemeine Parameter als Ergebnis liefert. Ein Element z liegt genau dann im Wertebereich von F_{gs}, wenn es entweder schon in der Lösungsmenge *solution* liegt, oder es liegt in der durch einen Deskriptor implizit beschriebenen Menge, die zur aktiven Menge gehört, und z erfüllt die Ein-/Ausgabedingung O. Da in F_{gs} Rekursion auftritt, brauchen wir ein adäquates Induktionsschema, um *lemma*1 und *lemma*11 beweisen zu können. Hierzu eignet sich die sogenannte *Measure-Induktion*, welche auch als *wohlfundierte Induktion* bzw. *noethersche Induktion* bezeichnet wird. Dieses Induktionsschema ist in PVS als Axiom (für wohlfundierte

[6] Aus Darstellungsgründen haben wir durchgehend den Typ *Prop* anstatt *Bool* verwendet.

```
% Subtype TCC generated for x
  % proved - complete
F_gs_TCC1:
  OBLIGATION
    (FORALL (aktiv: set[S]), (solution: set[R]),
     (x: {y: D | invariant(aktiv, solution, y)}):
       NOT empty?[S](aktiv) IMPLIES
         invariant(union[S](proj_2(arbsplit(aktiv)),
                            split(x, proj_1(arbsplit(aktiv)))),
                   union[R](solution, extract(x, proj_1(arbsplit(aktiv)))), x))

% Termination TCC generated for F_gs
  % proved - complete
F_gs_TCC2:
  OBLIGATION
    (FORALL (aktiv: set[S]), (solution: set[R]),
    (x: {y: D | invariant(aktiv, solution, y)}):
       NOT empty?(aktiv) IMPLIES
          kard(huelle(union(proj_2(arbsplit(aktiv)),
                            split(x, proj_1(arbsplit(aktiv)))), x))
           < kard(huelle(aktiv, x)))

% Subtype TCC generated for x
  % proved - complete
korrekt_TCC1: OBLIGATION (FORALL (x: {x1: D | I(x1)}):
  invariant(add[S](init(x), emptyset[S]), emptyset[R], x))
```

Abbildung 7: Typkorrektheitsbedingungen

Datentypen) vordefiniert. Das allgemeine Schema lautet (in QED Syntax)

$$
\begin{aligned}
&\mathit{induct\_scheme} :=\\
&\quad \forall\, T \mid \mathit{Type},\ \mathit{measure} : T \to \mathbb{N}, P : T \to \mathit{Prop}.\\
&\quad (\forall\, x : T.\ (\forall\, y : T.\ \mathit{measure}(y) < \mathit{measure}(x) \Rightarrow P(y)) \Rightarrow P(x)) \Rightarrow\\
&\quad \forall\, x : T.\ P(x)
\end{aligned}
$$

Als Maß für diese Induktion bietet es sich hier an, dasselbe wie bei der Definition von F_{gs} zu verwenden, nämlich die Kardinalität der transitiven Hülle der aktiven Menge. Wir haben dieses Schema für unser Problem bereits instanziiert und in die PVS-Theorie mit aufgenommen. Die Axiome *me_induct* und *me_induct*1 stellen dieses instanziierte Schema dar, wobei p genau die Aussage von *lemma*1, $p1$ die Aussage von *lemma*11 und *me* das in der Maßfunktion von F_{gs} definierte Maß darstellen. Weiterhin benötigen wir für den Beweis von *lemma*1 und *lemma*11 eine Beziehung zwischen den Prädikaten *satisfies* und O, welche durch das *lemma*2 beschrieben wird. Außerdem benötigen wir die folgenden Korollare:

- *unique_father* : im wesentlichen eine Folgerung der Tatsache, daß jeder Knoten höchstens einen Vorgänger hat ($ax8$). Dieses Korollar wird für den Beweis der Gültigkeit der Invariante benötigt, die mit den Parametern des rekursiven Aufrufs instanziiert ist.
- *tcl_sat_J* : Die Eigenschaft J (gültige Deskriptoren) wird durch die transitive Hülle vererbt.

- *satax* : ein Element, welches nicht aus der Menge r extrahiert werden kann, liegt in einem Nachfolger s von r.
- $satax_2$: ein aus der Menge s extrahiertes Element liegt in s.
- h_1 : kein Knoten s ist Nachfolger von sich selbst (eine Folgerung von $ax7$).
- $arb1, arb2$ beschreiben die Funktion *arbsplit* und folgen sofort aus der Axiomatisierung von *arbsplit*.

Um die korrekte Typisierung aller Entitäten zu gewährleisten, werden vom System (automatisch) sogenannte *Typkorrektheitsbedingungen* generiert. Die von PVS generierten Typkorrektheitsbedingungen sind in der Abb. 7 dargestellt. Die Bedingungen F_gs_TCC1 und $korrekt\_TCC1$ beschreiben die Anforderung, daß sowohl die Initialparameter als auch die Parameter des rekursiven Aufrufs die Invariante erfüllen müssen. Der Beweis von F_gs_TCC1 ist dabei am aufwendigsten, insbesondere wegen der dritten Bedingung der Invariante. Um diese nachzuweisen, braucht man im wesentlichen die Eigenschaften einer Baumstruktur wie z.B. Zyklenfreiheit.

Desweiteren müssen wir den Nachweis der Terminierung von F_{gs} erbringen. Die Beweisverpflichtung hierzu ergibt sich sofort aus der Typkorrektheit von F_{gs}, d. h. wir müssen zeigen, daß das durch die *measure*-Funktion definierte Maß bei jedem rekursiven Aufruf von F_{gs} kleiner wird. F_gs_TCC2 stellt genau diese Beweisverpflichtung dar, die wir mit Hilfe der Lemmata $termax_1$ und $termax_2$ zeigen.

Zum Schluß bleibt noch zu zeigen, daß F_{gs} in der Tat eine konstruktive Lösung der Anforderungsspezifikation *req_spec* ist. Der Beweis von *req_spec* ergibt sich sofort aus dem Korrektheitslemma *korrekt* und der Typkorrektheitsbedingung $korrekt\_TCC1$, wobei wir für den Existenzquantor in *req_spec* gerade den auf die Initialwerte angewandten schematischen Algorithmus F_{gs} einsetzen.

Alle Lemmata, Korollare und die sich aus der Typkorrektheit ergebenen Beweisverpflichtungen wurden erfolgreich mit Hilfe des PVS-Beweisers bewiesen. Die vollständigen Beweise in Form von Beweisskripten sind beim Autor erhältlich. Als Beispielbeweis ist im Anhang C der Beweis des Lemmas *lemma2* aufgeführt. Wir gehen in diesem Bericht nicht näher auf die verschiedenen Beweiskommandos des PVS Beweisers ein. Nähere Informationen, insbesondere auch über die Verwendung von Taktiken findet man in [12].

7 Ein Beispiel

In diesem Abschnitt wollen wir anhand eines einfachen Beispiels zeigen, wie man den in den vorherigen Abschnitten formalisierten Entwicklungsschritt der globalen Suche anwenden kann. Es wird sich dabei herausstellen, daß die Problemformalisierung wenig Schwierigkeiten bereitet, die Realisierung der Mengendeskriptoren und der Operationen auf dieser Datenstruktur aber mehr Ideen erfordert.

Betrachten wir nun folgendes Suchproblem *Key_search*:
Gegeben sei:

- eine natürliche Zahl n
- eine (monotone) Funktion A auf $\{1 \cdots n\}$ und

$d_p := \ll n : \mathbb{N}_1,\ A : \{n_1 : \mathbb{N}_1 \mid n_1 \leq n\} \to \mathbb{N},\ key : Nat \gg$
$r_p := \mathbb{N}$
$i_p := \lambda\ (n, A, key) : d_p.\ monoton(A)$
$o_1 := \lambda\ ((n, A, key), index) : d_p \times r_p.\ ((1 \leq index \leq n) \wedge (A(index) = key))$
$(d_p, r_p, i_p, o_p) :: Problemspec$
$Key\_search := req\_spec((d_p, r_p, i_p, o_p))$

Abbildung 8: Anforderungsspezifikation von *Key_search*

- ein Schlüsselelement *key*.

Gesucht ist nun:

- die Menge aller 'Indizes' zwischen 1 und n, so daß für jeden Index gilt: $A(index) = key$.

Zunächst müssen wir die Anforderungsspezifikation des Problems in Form einer *Problemspec*, d.h. als ein Viertupel formalisieren. Als Problemdomain ergibt sich offensichtlich ein Tripel. Als Eingabebedingung formalisieren wir die Anforderung an A, daß es sich um eine monotone Funktion handelt. Wir könnten diese Bedingung auch als Subtypbedingung im Problemdomain formalisieren. Die Eingabebedingung würde sich dann zu *true* vereinfachen. Prinzipiell kann man durch Verwendung von semantischen Subtypen immer auf die Eingabededingung verzichten, erspart sich dadurch aber nicht die Arbeit, diese bei Instanziierung nachzuweisen. Der Ausgabebereich des Problems sind die natürlichen Zahlen. Wir erhalten die in Abb. 8 angegebene Anforderungsspezifikation. Wir haben nun das Problem eindeutig spezifiziert. Um die *Globale Suche* anwenden zu können, benötigen wir eine Erweiterung der Anforderungsspezifikation des Problems zu einer globalen Suchtheorie. Die Idee ist nun, das Verfahren der binären Suche durchzuführen, wobei die Suche in einem Anfangsintervall beginnt, welches dann solange in zwei etwa gleichgroße Intervalle aufgeteilt wird, bis nur noch einelementige Intervalle vorhanden sind. Den Typ der Deskriptoren repräsentieren wir somit als ein Tupel, welches genau die untere und obere Grenze des Intervalls darstellt. Das Anfangsintervall umfaßt den gesamten Suchbereich, in diesem Fall den Bereich zwischen 1 und n. Gültige Intervalle (in Abhängigkeit der Eingabe) sind nun alle Intervalle, die zwischen 1 und n liegen. Ein Intervall splittet genau in zwei gleiche Teilintervalle, falls es kein einelementiges Intervall ist. Potentielle Lösungen kann man direkt aus einelementigen Intervallen extrahieren. Wir erhalten die in der Abb. 9 dargestellte Realisierung. Die Funktionen *split* und *extract* ergeben sich in naheliegender Weise aus den Prädikaten *split*? bzw. *extract*?. *split* liefert zu einem gegebenen Intervall genau die zwei Teilintervalle und *extract* gibt zu einem einelementigen Intervall genau diesen Wert zurück, falls die Bedingung O erfüllt ist. Das *Typcasting* (symbolisiert durch ::) fordert, daß diese Struktur in der Tat eine Realisierung einer globalen Suchtheorie darstellt. Dadurch werden eine Reihe von Beweisverpflichtungen generiert. Dies sind gerade die mit den aktuellen Werten d_p, r_p usw. instanziierten Axiome der *global_search_theory*. Die meisten Beweisverpflichtungen sind trivial. Beispielsweise ergibt sich das dritte Axiom ($ax3$) durch Instanziierung zu:

$$Goal_3 : \forall (n, A, key) : d_p, z : \mathbb{N}.\ (monoton(A) \wedge (1 \leq z \leq n) \wedge (A(z) = key)) \Rightarrow (1 \leq z \leq n)$$

Sind alle Beweisverpflichtungen bewiesen, so erhält man sofort eine korrekte Lösung f_{key_search}

$key\_search\_theory :=$

STRUC

$S := \ll i : \mathbb{N}, j : \mathbb{N} \gg$

$J := \lambda\ ((n, A, key), (i, j)) : d_p \times S.\ (1 \leq i \leq j \leq n)$

$init := \lambda\ ((n, A, key), (i, j)) : d_p \times S.\ (1, n)$

$satisfies := \lambda\ (k, (i, j)) : r_p \times S.\ (i \leq k \leq j)$

$split? := \lambda\ (n, A, key) : d_p,\ (i_1, j_1), (i_2, j_2) : S.$
$(i_1 < j_1) \wedge$
$(i_2, j_2) = (i_1, (i_1 + j_1) \text{ div } 2) \vee$
$(i_2, j_2) = (1 + (i_1 + j_1) \text{ div } 2, j_1)$

$split := \lambda\ ((n, A, key), (i, j)) : d_p \times S.$
IF $(i < j)$ THEN
LET
$(i_1, j_1) = (i, (i + j) \text{ div } 2),$
$(i_2, j_2) = (1 + (i + j) \text{ div } 2, j)$
IN
$insert((i_1, j_1), insert((i_2, j_2), \varnothing))$
ELSE $\varnothing$

$extract? := \lambda\ (k, (i, j)) : r_p \times S.\ ((i = j) \wedge (k = i))$

$extract := \lambda\ ((n, A, key), (i, j)) : d_p \times S.$
IF $(i = j) \wedge (A(i) = key)$ THEN $insert(i, \varnothing)$
ELSE $\varnothing$

END $:: global\_search\_theory\ (d_p, r_p, i_p, o_p)$

Abbildung 9: Die *Globale Suchtheorie* für die binäre Suche

$$
\begin{array}{l}
f := F_{gs}((d_p, r_p, i_p, o_p), key\_search\_theory, arbsplit, huelle) \\
\quad = \ \text{FIX } f : F\_Type \to Set(r_p). \\
\qquad \lambda\ (aktiv, solution, x) : F\_Type. \\
\qquad \text{IF } empty?(aktiv) \text{ THEN } solution \\
\qquad \text{ELSE} \\
\qquad \text{LET} \\
\qquad\quad ((i,j), A_1) := arbsplit(aktiv), \\
\qquad\quad Newaktiv := A_1 \cup \\
\qquad\qquad\qquad\qquad \text{IF } (i < j) \text{ THEN} \\
\qquad\qquad\qquad\qquad \text{LET} \\
\qquad\qquad\qquad\qquad\quad (i_1, j_1) = (i, (i+j) \text{ div } 2), \\
\qquad\qquad\qquad\qquad\quad (i_2, j_2) = (1 + (i+j) \text{ div } 2, j) \\
\qquad\qquad\qquad\qquad \text{IN} \\
\qquad\qquad\qquad\qquad\quad insert((i_1, j_1), insert((i_2, j_2), \varnothing)) \\
\qquad\qquad\qquad\qquad \text{END} \\
\qquad\qquad\qquad\qquad \text{ELSE } \varnothing \\
\qquad\quad Newsolution := solution \cup \\
\qquad\qquad\qquad\qquad\quad \text{IF } (i = j) \wedge (A(i) = key) \text{ THEN } insert(i, \varnothing) \\
\qquad\qquad\qquad\qquad\quad \text{ELSE } \varnothing \\
\qquad \text{IN} \\
\qquad\quad f(Newaktiv, Newsolution, x) \\
\qquad \text{END} \\
f\_key\_search := \\
\qquad \lambda\ (n, A, key) : \{d_p \mid i_p((n, A, key))\}.\ f(insert((1, n), \varnothing), \varnothing, (n, A, key))
\end{array}
$$

Abbildung 10: Lösung des *Key_Search-Problems*

des Problems durch einfache Instanziierung des Schemas F_{gs}, wobei wir voraussetzen, daß im Kontext eine Funktion *arbsplit* gegeben ist, die jeweils ein Intervall in der aktiven Menge auswählt. Weiterhin setzen wir voraus, daß eine Funktion *huelle* im aktuellen Kontext deklariert ist. Wir erhalten die in Abb. 10 angegebene (Teil)-expandierte Lösung.

Die so erhaltene Lösung *f_key_search* läßt sich offensichtlich noch optimieren. Durch die Tatsache, daß die gegebene Funktion A monoton ist, brauchen nur diejenigen Intervalle (i, j) überprüft zu werden, für welche $A(i) \leq key \leq A(j)$ gilt. Durch Hinzunahme von sogenannten Filtern, und zusätzlicher Anwendung von optimierenden Transformationen, kann man den oben erhaltenen linearen Algorithmus so modifizieren, daß sich letztendlich ein Algorithmus mit logarithmischer Komplexität ergibt. Wir wollen in diesem Bericht nicht näher auf diese Dinge eingehen, Informationen hierüber findet man etwa in [18].

8 Zusammenfassung

Wir haben in diesem Bericht demonstriert, daß ein typtheoretischer Ansatz einen adäquaten Rahmen zur Verfügung stellt, einen „mächtigen“ generischen Software-Entwicklungsschritt zu formalisieren und dessen Korrektheit nachzuweisen. Die Formalisierung erfolgte dabei rein auf der Objektebene, also auf der Ebene von Spezifikationen und Datenstrukturen mittels Funktionen höherer Ordnung. Der Entwicklungsschritt *Globale Suche* wurde durch eine Funktion höherer Ordnung repräsentiert, welche angewandt auf eine *global_search_theory* eine Problemlösung lieferte, wobei die *global_search_theory* als eine Erweiterung des ursprünglichen Problems um Datenstrukturen und Operationen auf diesen zu betrachten war. Hatte man ein Problem gegeben, so bestand die Hauptaufgabe darin, konkrete Realisierungen der abstrakten Datenstrukturen zu finden, so daß die Axiome der globalen Suchtheorie erfüllt waren. Wir sahen, daß sich im Formalismus QED durch Typkorrektheitsbedingungen diese Beweisverpflichtungen in natürlicher Weise ergaben. Die meisten davon ließen sich sehr einfach beweisen oder waren trivial, nur wenige erforderten ein diffizileres Beweisvorgehen. Hat man nun alle diese Beweisverpflichtungen bewiesen, so konnte man durch Funktionsapplikation, also durch Instanziierung des schematischen Algorithmus, sofort eine Lösung angeben. Die maschinelle Beweisunterstützung erfolgte mit Hilfe des Spezifikations- und Beweissystems PVS, welches einen Gentzen-Beweiser für Logik höherer Ordnung zur Verfügung stellt.

Entwicklungsschritte können allgemein als Transformationen betrachtet werden, die angewandt auf eine Spezifikation eine neue Spezifikation liefern, welche eine Verfeinerung der ursprünglichen darstellt. Solche Transformationen erfordern jedoch gelegentlich den Zugriff auf die innere syntaktische Struktur einer Spezifikation, um diese in geeigneter Form zu modifizieren, beispielsweise die Ersetzung eines Axiomes durch ein stärkeres oder die Hinzufügung zusätzlicher Datenstrukturen zu einer Spezifikation. Transformationen dieser Art müssen somit auf syntaktischen Repräsentationen von Spezifikationen operieren. Spezifikationen und Transformationen auf diesen Spezifikationen lassen sich deshalb im allg. nicht im selben (Objekt-)Formalismus repräsentieren und nachweisen, sondern erfordern die Einbeziehung eines zusätzlichen (Meta-) Formalismus [1]. Hierzu gehören insbesondere auch Transformationen, die innerhalb einer Spezifikation eine Funktion optimieren. Die Sprache QED stellt hinreichend mächtige Konzepte zur Verfügung, um die Meta-Theorie von QED in sich selbst zu reflektieren [22].

Die in diesem Bericht beschriebene Vorgehensweise läßt sich ohne weiteres sowohl auf mächtige generische Entwicklungsschritte der hier beschriebenen Art (wie z. B. auch *divide-and-conquer*) als auch auf jene Transformationen wie sie etwa im CIP-Projekt [3, 4] betrachtet wurden, z. B. die Operationalisierung eines Existenzquantors, anwenden. Die langfristige Zielsetzung hiebei ist der Aufbau einer Bibliothek von Entwicklungsschritten unterschiedlicher

Komplexität und Art (sowohl höhere Funktionale als auch Meta-Operatoren) als Basis für ein Entwicklungssystem für korrekte und sichere Software.

Literatur

[1] David A. Basin and Robert L. Constable. Metalogical frameworks. Technical Report TR 91-1235, Cornell University, Ithaca, New York, September 1991.

[2] Robert L. Constable et. al. *Implementing Mathematics with the NuPRL proof development system.* Prentice Hall, Englewood Cliffs, NJ, 1986.

[3] The CIP Language Group. *The Munich Project CIP, Volume I: The Wide Spectrum Language CIP-L.* LNCS 183. Springer-Verlag, 1985.

[4] The CIP System Group. *The Munich Project CIP - Volume II: The Program Transformation System CIP-S.* LNCS 292. Springer-Verlag, 1987.

[5] W.A. Howard. The Formulae-as-Types Notion of Construction. In J. Hindley and J. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, 1980.

[6] G. Huet and B. Lang. Proving and applying program transformations expressed with second-order-patterns. *Acta Informatica*, 11:31–55, 1978.

[7] C. Kreitz. Metasynthesis - deriving programs that develop programs. Technical Report AIDA-93-03, Fachgebiet Intellektik, Technische Hochschule Darmstadt, 1993.

[8] Z. Luo. An Extended Calculus of Constructions. Technical Report CST-65-90, University of Edinburgh, July 1990.

[9] Z. Luo. Program Specification and Data Refinement in Type Theory. In S. Abramsky and T.S.E. Maibaum, editors, *TAPSOFT'91, volume I*, pages 143–168. Springer-Verlag, LNCS 493, 1991.

[10] Ch.E. Ore. The extended calculus of constructions (ECC) with inductive types. *Information and Computation*, 99:231–264, 1992.

[11] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, 1992. Springer-Verlag.

[12] S. Owre, N. Shankar, and J.M. Rushby. *The PVS Proof Checker: A Reference Manual.* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[13] H.A. Partsch. *Specification and Transformation of Programs.* Springer-Verlag, 1990.

[14] H. Rueß. Report on the specification language QED. Korso paper, Universität Ulm, 1993.

[15] Donald Sannella and Andrzej Tarlecki. Extended ML: Past, present and future. In H. Ehrig et al., editor, *Recent Trends in Data Type Specifications*, pages 297 – 322. Springer LNCS 534, 1991.

[16] D. Schwier. Type checking the specification language QED. Korso paper, Universität Ulm, 1993.

[17] Douglas R. Smith. Applications of a strategy for designing divide-and-conquer-algorithms. *Science of Computer Programming*, (8):213–229, 1987.

[18] Douglas R. Smith. Structure and design of global search algorithms. Technical Report KES.U.87.12, Kestrel Institute, Palo Alto, CA, 1987.

[19] Douglas R. Smith. *KIDS - A knowledge-based Sofware Development System*, chapter to appear. Automating Software Design. Live Oak Press, Menlo Park, CA, 1991.

[20] Douglas R. Smith and Michael R. Lowry. Algorithm theories and design tactics. *Science of Computer Programming*, (14):305–321, 1990.

[21] Simon Thompson. *Type Theory And Functional Programming.* International Computer Science Series. Addison Wesley, 1991.

[22] F.W. von Henke, A. Dold, M. Grosse, H. Rueß, D. Schwier, and M. Strecker. Construction and deduction methods for the formal development of software. In *Arbeitsberichte des* KORSO-*Projekts*. Springer LNCS, 1994. Erscheint im Laufe des Jahres; Titel des Bandes noch unsicher.

[23] Martin Wirsing. A framework for software development in korso: Results of the meeting of the korso methodic group. Technical Report 92-05, Ludwig-Maximilians-University Munich, 1992.

A PVS-Spezifikation der Globalen Suche

```
global_search[D : TYPE, R : TYPE, I : [D → bool], O : [D, R → bool], S : TYPE,
              J : [D, S → bool], init : [D → S],
              satisfies : [R, S → bool], split? : [D, S, S → bool],
              split : [D, S → set[S]], extract? : [R, S → bool],
              extract : [D, S → set[R]],
              arbsplit : [set[S] → [S, set[S]]], huelle : [set[S], D → set[S]]] :
 THEORY
  BEGIN

   ASSUMING
    IMPORTING transclos3[S, D, split?]
    ax1 : ASSUMPTION (∀ (x : D) : I(x) ⊃ J(x, init(x)))

    ax2 :
     ASSUMPTION
      (∀ (x : D), (r, s : S) :
        (I(x) ∧ J(x, r) ∧ (split?(x, r, s))) ⊃ J(x, s))

    ax3 :
     ASSUMPTION
      (∀ (x : D), (z : R) : (I(x) ∧ O(x, z)) ⊃ satisfies(z, init(x)))

    ax4 :
     ASSUMPTION
      (∀ (x : D), (r : S), (z : R) :
        (I(x) ∧ J(x, r))
          ⊃
         (satisfies(z, r)
             ⇔
           (extract?(z, r)
               ∨ (∃ (s : S) : (transclosure?(x, r, s) ∧ extract?(z, s))))))

    ax5 :
     ASSUMPTION
      (∀ (x : D), (r : S) :
        (I(x) ∧ J(x, r))
          ⊃
         (∀ (z : R) :
           z ∈ extract(x, r) ⇔ (extract?(z, r) ∧ O(x, z))))

    ax6 :
     ASSUMPTION
      (∀ (x : D), (r : S) :
        (I(x) ∧ J(x, r))
          ⊃ (∀ (s : S) : (s ∈ split(x, r) ⇔ split?(x, r, s))))

    ax7 : ASSUMPTION (∀ (x : D), (s : S) : ¬ (transclosure?(x, s, s)))

    ax8 :
     ASSUMPTION
      (∀ (x : D), (r1, r2 : S), (s : S) :
        (I(x) ∧ J(x, r1) ∧ J(x, r2))
          ⊃ (split?(x, r1, s) ∧ split?(x, r2, s)) ⊃ r1 = r2)
```

```
  arbsplitax1 :
    ASSUMPTION
      (∀ (m : set[S]) :
        ¬ (empty?[S](m))
          ⊃ m = add(proj_1(arbsplit(m)), proj_2(arbsplit(m))))

  arbsplitax2 :
    ASSUMPTION
      (∀ (m : set[S]) :
        ¬ (empty?[S](m))
          ⊃ ¬ proj_1(arbsplit(m)) ∈ proj_2(arbsplit(m)))

  huelle_ax :
    ASSUMPTION
      (∀ (aktiv : set[S]), (x : D) :
        (∀ (t : S) :
          (t ∈ huelle(aktiv, x)
              ⇔
            ((∃ (r : S) : (r ∈ aktiv ∧ transclosure?(x, r, t)))
                ∨ t ∈ aktiv))))

    ENDASSUMING

kard : [set[S] → nat]
kardax1 : AXIOM kard(emptyset[S]) = 0

kardax2 :
  AXIOM
    (∀ (s_1, s_2 : set[S]) :
      (subset?(s_1, s_2) ∧ ¬ (subset?(s_2, s_1))) ⊃ kard(s_1) < kard(s_2))

invariant(aktiv : set[S], solution : set[R], x : D) :
  bool
      =
      I(x)
        ∧ (∀ (s : S) : (s ∈ aktiv ⊃ J(x, s)))
          ∧ (∀ (z : R) : (z ∈ solution ⊃ O(x, z)))
            ∧
            (∀ (s_1, s_2 : S) :
              ((s_1 ∈ aktiv ∧ s_2 ∈ aktiv) ⊃ ¬ (transclosure?(x, s_1, s_2))))

F_gs(aktiv : set[S], solution : set[R], x : {y : D | invariant(aktiv, solution, y)}) :
  RECURSIVE set[R] =
    IF empty?(aktiv) THEN solution
      ELSE LET
              newaktiv : set[S] =
                union(proj_2(arbsplit(aktiv)), split(x, proj_1(arbsplit(aktiv)))),
              newsolution : set[R] =
                union(solution, extract(x, proj_1(arbsplit(aktiv))))
        IN F_gs(newaktiv, newsolution, x)
      ENDIF
  MEASURE
    (λ (aktiv : set[S]), (solution : set[R]),
       (x : {y : D | invariant(aktiv, solution, y)}) :
```

kard(huelle(aktiv, x)))

termax :
LEMMA
(∀ (aktiv : set[S]), (solution : set[R]),
(x : {y : D | invariant(aktiv, solution, y)}) :
¬ (empty?[S](aktiv))
⊃
subset?
(huelle(union(proj_2(arbsplit(aktiv)), split(x, proj_1(arbsplit(aktiv)))),
x),
huelle(aktiv, x)))

termax2 :
LEMMA
(∀ (aktiv : set[S]), (solution : set[R]),
(x : {y : D | invariant(aktiv, solution, y)}) :
¬ (empty?[S](aktiv))
⊃ ¬
(subset?(huelle(aktiv, x),
huelle(union(proj_2(arbsplit(aktiv)),
split(x, proj_1(arbsplit(aktiv)))),
x))))

F_Type :
TYPE =
[aktiv : set[S], solution : set[R], x : {y : D | invariant(aktiv, solution, y)}]
me(aktiv : set[S], solution : set[R], x : {y : D | invariant(aktiv, solution, y)}) :
nat = kard(huelle(aktiv, x))

p(aktiv : set[S], solution : set[R], x : {y : D | invariant(aktiv, solution, y)}) :
bool
=
(∀ (z : R) :
z ∈ F_gs(aktiv, solution, x)
⊃
((z ∈ solution
∨ (∃ (r : S) : (r ∈ aktiv ∧ satisfies(z, r) ∧ O(x, z))))))

p_1(aktiv : set[S], solution : set[R], x : {y : D | invariant(aktiv, solution, y)}) :
bool
=
(∀ (z : R) :
((z ∈ solution
∨ (∃ (r : S) : (r ∈ aktiv ∧ satisfies(z, r) ∧ O(x, z)))))
⊃ z ∈ F_gs(aktiv, solution, x))

me_induct :
AXIOM
(∀ (aktiv1 : set[S]), (solution1 : set[R]),
(x_1 : {y : D | invariant(aktiv1, solution1, y)}) :
(∀ (aktiv2 : set[S]), (solution2 : set[R]),
(x_2 : {y : D | invariant(aktiv2, solution2, y)}) :
me(aktiv2, solution2, x_2) < me(aktiv1, solution1, x_1)
⊃ p(aktiv2, solution2, x_2))

$\supset\ p(\text{aktiv1},\ \text{solution1}, x_1))$
$\supset$
$(\forall\ (\text{aktiv1} :\ \text{set}[S]), (\text{solution1} :\ \text{set}[R]),$
$(x_1 :\ \{y :\ D\ \mid\ \text{invariant}(\text{aktiv1},\ \text{solution1}, y)\}) :$
$p(\text{aktiv1},\ \text{solution1}, x_1))$

me_induct1 :
AXIOM
$(\forall\ (\text{aktiv1} :\ \text{set}[S]), (\text{solution1} :\ \text{set}[R]),$
$(x_1 :\ \{y :\ D\ \mid\ \text{invariant}(\text{aktiv1},\ \text{solution1}, y)\}) :$
$(\forall\ (\text{aktiv2} :\ \text{set}[S]), (\text{solution2} :\ \text{set}[R]),$
$(x_2 :\ \{y :\ D\ \mid\ \text{invariant}(\text{aktiv2},\ \text{solution2}, y)\}) :$
$\text{me}(\text{aktiv2},\ \text{solution2}, x_2)\ <\ \text{me}(\text{aktiv1},\ \text{solution1}, x_1)$
$\supset\ p_1(\text{aktiv2},\ \text{solution2}, x_2))$
$\supset\ p_1(\text{aktiv1},\ \text{solution1}, x_1))$
$\supset$
$(\forall\ (\text{aktiv1} :\ \text{set}[S]), (\text{solution1} :\ \text{set}[R]),$
$(x_1 :\ \{y :\ D\ \mid\ \text{invariant}(\text{aktiv1},\ \text{solution1}, y)\}) :$
$p_1(\text{aktiv1},\ \text{solution1}, x_1))$

unique_father :
COROLLARY
$(\forall\ (x :\ D), (r, q, s :\ S) :$
$(I(x)\ \wedge\ J(x, r)\ \wedge\ J(x, q)\ \wedge\ (q\ \neq\ r)\ \wedge\ \text{split?}(x, r, s)\ \wedge\ \text{transclosure?}(x, q, s))$
$\supset\ \text{transclosure?}(x, q, r))$

tcl_sat_J :
COROLLARY
$(\forall\ (x :\ D), (r, s :\ S) :\ I(x)\ \wedge\ J(x, r)\ \wedge\ \text{transclosure?}(x, r, s)\ \supset\ J(x, s))$

satax :
COROLLARY
$(\forall\ (x :\ D), (r, s :\ S), (z :\ R) :$
$((I(x)\ \wedge\ J(x, r)\ \wedge\ \neg\ (\text{extract?}(z, r)))$
$\supset\ ((\text{split?}(x, r, s)\ \wedge\ \text{satisfies}(z, s))\ \supset\ \text{satisfies}(z, r))))$

satax2 :
COROLLARY
$(\forall\ (x :\ D), (s :\ S), (z :\ R) :$
$((I(x)\ \wedge\ J(x, s))\ \supset\ (\text{extract?}(z, s)\ \supset\ \text{satisfies}(z, s))))$

h_1 : COROLLARY $(\forall\ (x :\ D), (s :\ S) :\ \neg\ (\text{split?}(x, s, s)))$

arb1 : COROLLARY $(\forall\ (m :\ \text{set}[S]) :\ \neg\ (\text{empty?}[S](m))\ \supset\ \text{proj\_1}(\text{arbsplit}(m)) \in m)$

arb2 :
COROLLARY
$(\forall\ (m :\ \text{set}[S]), (s :\ S) :$
$\neg\ (\text{empty?}[S](m))\ \supset\ s \in \text{proj\_2}(\text{arbsplit}(m))\ \supset\ s \in m)$

lemma1 :
LEMMA
$(\forall\ (\text{aktiv} :\ \text{set}[S]), (\text{solution} :\ \text{set}[R]),$
$(x :\ \{y :\ D\ \mid\ \text{invariant}(\text{aktiv},\ \text{solution}, y)\}), (z :\ R) :$
$z \in \text{F\_gs}(\text{aktiv},\ \text{solution}, x)$

$\supset$
((z ∈ solution
∨ (∃ (r : S) : (r ∈ aktiv ∧ satisfies(z, r) ∧ O(x, z))))))

lemma11 :
LEMMA
(∀ (aktiv : set[S]), (solution : set[R]),
(x : {y : D | invariant(aktiv, solution, y)}), (z : R) :
((z ∈ solution
∨ (∃ (r : S) : (r ∈ aktiv ∧ satisfies(z, r) ∧ O(x, z)))))
$\supset$ z ∈ F_gs(aktiv, solution, x))

lemma2 :
LEMMA
(∀ (aktiv : set[S]), (solution : set[R]),
(x : {y : D | invariant(aktiv, solution, y)}),
(z : R), (s : {s_1 : S | s_1 ∈ aktiv}) :
(satisfies(z, s) ∧ O(x, z))
=
((extract?(z, s) ∧ O(x, z))
∨ (∃ (s_1 : S) : split?(x, s, s_1) ∧ satisfies(z, s_1) ∧ O(x, z))))

korrekt :
THEOREM
(∀ (x : {x_1 : D | I(x_1)}), (z : R) :
z ∈ F_gs(add(init(x), emptyset[S]), emptyset[R], x) ⇔ O(x, z))

req_spec :
THEOREM
(∀ (x : {x_1 : D | I(x_1)}) :
(∃ (s : set[R]) : (∀ (elem : R) : (elem ∈ s ⇔ O(x, elem)))))

END global_search

B PVS-Spezifikation der transitiven Hülle

```
transclos3[T : TYPE, D : TYPE, Rel : [D, T, T → bool]] : THEORY
  BEGIN

  trans?(R : [D, T, T → bool]) :
    bool = ∀ (x, y, z : T), (d : D) : R(d, x, y) ⊃ R(d, y, z) ⊃ R(d, x, z)

  hered?(P : [D, T → bool], d : D) :
    bool = ∀ (x, y : T) : P(d, x) ⊃ Rel(d, x, y) ⊃ P(d, y)

  transclosure?(d : D, x, y : T) :
    bool
      =
      ∀ (P : [D, T → bool]) :
        hered?(P, d) ⊃ (∀ (u : T) : Rel(d, x, u) ⊃ P(d, u)) ⊃ P(d, y)

  trans_is_transitiv : THEOREM trans?(transclosure?)

  minimality :
    THEOREM
      (∀ (R : [D, T, T → bool]) :
        trans?(R)
          ⊃ (∀ (x, y : T), (d : D) : Rel(d, x, y) ⊃ R(d, x, y))
            ⊃ (∀ (x, y : T), (d : D) : transclosure?(d, x, y) ⊃ R(d, x, y)))

  rel_tcl :
    LEMMA (∀ (d : D), (x, y : T) : Rel(d, x, y) ⊃ transclosure?(d, x, y))

  hilfslemma :
    THEOREM
      (∀ (d : D), (x, y : T) :
        (transclosure?(d, x, y)
          =
          ((∃ (z : T) : (Rel(d, x, z) ∧ transclosure?(d, z, y)))
            ∨ Rel(d, x, y))))

  hilfslemma2 :
    LEMMA
      (∀ (d : D), (x, y : T) :
        (transclosure?(d, x, y)
          =
          (Rel(d, x, y)
            ∨ (∃ (z : T) : transclosure?(d, x, z) ∧ Rel(d, z, y)))))

  END transclos3
```

C Beispielbeweis

Terse proof for `lemma2`.

`lemma2:`

$$
\begin{array}{ll}
\{1\} & (\forall\ (\text{aktiv}:\ \text{set}[S]), (\text{solution}:\ \text{set}[R]),\\
& \quad (x:\ \{y:\ D\ \mid\ \text{invariant}(\text{aktiv},\ \text{solution}, y)\}),\\
& \quad (z:\ R), (s:\ \{s_1:\ S\ \mid\ s_1 \in \text{aktiv}\}):\\
& \quad (\text{satisfies}(z, s)\ \wedge\ O(x, z))\\
& \qquad =\\
& \quad ((\text{extract?}(z, s)\ \wedge\ O(x, z))\\
& \qquad \vee\ (\exists\ (s_1:\ S):\ \text{split?}(x, s, s_1)\ \wedge\ \text{satisfies}(z, s_1)\ \wedge\ O(x, z))))
\end{array}
$$

For the top quantifier in 1, we introduce Skolem constants: (aktiv!1 solution!1 x' z' s'),

Applying ax4 where

Instantiating the top quantifier in -1 with the terms: (x' s' z'),

Adding type constraints for x',

Expanding the definition of invariant,

Applying disjunctive simplification to flatten sequent,

Adding type constraints for s',

Instantiating the top quantifier in -3 with the terms: s',

Deleting some formulas,

Simplifying with decision procedures,

Invoking decision procedures,

Replacing using formula -5,

Deleting some formulas,

Invoking decision procedures,

Converting equality to IFF,

Splitting conjunctions,

we get 2 subgoals:

`lemma2.1:`

$$
\begin{array}{ll}
\{-1\} & s' \in \text{aktiv!1}\\
\{-2\} & I(x')\\
\{-3\} & J(x', s')\\
\{-4\} & (\forall\ (s_1, s_2:\ S):\\
& \quad ((s_1 \in \text{aktiv!1}\ \wedge\ s_2 \in \text{aktiv!1})\\
& \qquad \supset\ \neg\ (\text{transclosure?}(x', s_1, s_2))))\\
\hline
\{1\} & ((\text{extract?}(z', s')\\
& \quad \vee\ (\exists\ (s:\ S):\ (\text{transclosure?}(x', s', s)\ \wedge\ \text{extract?}(z', s))))\\
& \quad \wedge\ O(x', z'))\\
& \quad \supset\\
& \quad ((\text{extract?}(z', s')\ \wedge\ O(x', z'))\\
& \qquad \vee\ (\exists\ (s_1:\ S):\ \text{split?}(x', s', s_1)\ \wedge\ \text{satisfies}(z', s_1)\ \wedge\ O(x', z')))
\end{array}
$$

Applying disjunctive simplification to flatten sequent,

Splitting conjunctions,

we get 2 subgoals:

`lemma2.1.1:`

$$
\begin{array}{|ll}
\{\text{-}1\} & \text{extract?}(z', s') \\
\{\text{-}2\} & O(x', z') \\
 & \vdots \\
\hline
\{1\} & (\text{extract?}(z', s') \ \wedge \ O(x', z')) \\
\{2\} & (\exists \ (s_1 : \ S) : \ \text{split?}(x', s', s_1) \ \wedge \ \text{satisfies}(z', s_1) \ \wedge \ O(x', z'))
\end{array}
$$

Invoking decision procedures,

which is trivially true.

This completes the proof of `lemma2.1.1`.

`lemma2.1.2:`

$$
\begin{array}{|ll}
\{\text{-}1\} & (\exists \ (s : \ S) : \ (\text{transclosure?}(x', s', s) \ \wedge \ \text{extract?}(z', s))) \\
\{\text{-}2\} & O(x', z') \\
 & \vdots \\
\hline
\{1\} & (\text{extract?}(z', s') \ \wedge \ O(x', z')) \\
\{2\} & (\exists \ (s_1 : \ S) : \ \text{split?}(x', s', s_1) \ \wedge \ \text{satisfies}(z', s_1) \ \wedge \ O(x', z'))
\end{array}
$$

Applying transclos3.hilfslemma where

For the top quantifier in -2, we introduce Skolem constants: (s''),

Instantiating the top quantifier in -1 with the terms: $(x' \ s' \ s'')$,

Applying disjunctive simplification to flatten sequent,

Replacing using formula -1,

Splitting conjunctions,

we get 2 subgoals:

`lemma2.1.2.1:`

$$
\begin{array}{|ll}
\{\text{-}1\} & (\exists \ (z : \ S) : \ (\text{split?}(x', s', z) \ \wedge \ \text{transclosure?}(x', z, s''))) \\
\{\text{-}2\} & (\text{transclosure?}(x', s', s'') \\
 & \quad = \\
 & \quad ((\exists \ (z : \ S) : \ (\text{split?}(x', s', z) \ \wedge \ \text{transclosure?}(x', z, s''))) \\
 & \quad \ \vee \ \text{split?}(x', s', s''))) \\
\{\text{-}3\} & \text{extract?}(z', s'') \\
 & \vdots \\
\hline
 & \vdots
\end{array}
$$

For the top quantifier in -1, we introduce Skolem constants: (z''),

Instantiating the top quantifier in 2 with the terms: z'',

Invoking decision procedures,

Invoking decision procedures,

Invoking decision procedures,

Applying disjunctive simplification to flatten sequent,

Invoking decision procedures,

Applying ax4 where

Instantiating the top quantifier in -1 with the terms: $(x'\ z''\ z')$,

Applying ax2 where

Instantiating the top quantifier in -1 with the terms: $(x'\ s'\ z'')$,

Invoking decision procedures,

Invoking decision procedures,

Applying disjunctive simplification to flatten sequent,

Deleting some formulas,

Invoking decision procedures,

Applying disjunctive simplification to flatten sequent,

Instantiating the top quantifier in 2 with the terms: s'',

Invoking decision procedures,

which is trivially true.

This completes the proof of `lemma2.1.2.1`.

`lemma2.1.2.2`:

$$
\begin{array}{ll}
\{\text{-}1\} & \text{split?}(x', s', s'') \\
\{\text{-}2\} & (\text{transclosure?}(x', s', s'') \\
 & \quad = \\
 & ((\exists\ (z : S) :\ (\text{split?}(x', s', z)\ \wedge\ \text{transclosure?}(x', z, s''))) \\
 & \quad \vee\ \text{split?}(x', s', s''))) \\
\{\text{-}3\} & \text{extract?}(z', s'') \\
 & \vdots \\
\hline
 & \vdots
\end{array}
$$

Instantiating the top quantifier in 2 with the terms: s'',

Invoking decision procedures,

Applying satax where

Deleting some formulas,

Applying satax2 where

Invoking decision procedures,

Instantiating the top quantifier in -1 with the terms: $(x'\ s''\ z')$,

Applying ax2 where

Instantiating the top quantifier in -1 with the terms: $(x'\ s'\ s'')$,

Invoking decision procedures,

which is trivially true.

This completes the proof of `lemma2.1.2.2`.

`lemma2.2:`

$\{-1\}$ $s' \in \text{aktiv!1}$
$\{-2\}$ $I(x')$
$\{-3\}$ $J(x', s')$
$\{-4\}$ $(\forall\ (s_1, s_2 : S) :$
$((s_1 \in \text{aktiv!1}\ \wedge\ s_2 \in \text{aktiv!1})$
$\supset\ \neg\ (\text{transclosure?}(x', s_1, s_2))))$

$\{1\}$ $((\text{extract?}(z', s')\ \wedge\ O(x', z'))$
$\vee\ (\exists\ (s_1 : S) :\ \text{split?}(x', s', s_1)\ \wedge\ \text{satisfies}(z', s_1)\ \wedge\ O(x', z')))$
$\supset$
$((\text{extract?}(z', s')$
$\vee\ (\exists\ (s : S) :\ (\text{transclosure?}(x', s', s)\ \wedge\ \text{extract?}(z', s))))$
$\wedge\ O(x', z'))$

Invoking decision procedures,

Applying disjunctive simplification to flatten sequent,

Splitting conjunctions,

we get 2 subgoals:

`lemma2.2.1:`

$\{-1\}$ $(\text{extract?}(z', s')\ \wedge\ O(x', z'))$
$\vdots$

$\{1\}$ $((\text{extract?}(z', s')$
$\vee\ (\exists\ (s : S) :\ (\text{transclosure?}(x', s', s)\ \wedge\ \text{extract?}(z', s))))$
$\wedge\ O(x', z'))$

Invoking decision procedures,

Applying disjunctive simplification to flatten sequent,

Splitting conjunctions,

we get 2 subgoals:

`lemma2.2.1.1:`

$\{-1\}$ $\text{extract?}(z', s')$
$\{-2\}$ $O(x', z')$
$\vdots$

$\{1\}$ $(\text{extract?}(z', s')$
$\vee\ (\exists\ (s : S) :\ (\text{transclosure?}(x', s', s)\ \wedge\ \text{extract?}(z', s))))$

Applying disjunctive simplification to flatten sequent,

which is trivially true.

This completes the proof of `lemma2.2.1.1`.

`lemma2.2.1.2:`

$\{-1\}$ $\text{extract?}(z', s')$
$\{-2\}$ $O(x', z')$
$\vdots$

$\{1\}$ $O(x', z')$

which is trivially true.

This completes the proof of `lemma2.2.1.2`.

`lemma2.2.2:`

$$\begin{array}{ll} \{\text{-}1\} & (\exists\ (s_1 :\ S) :\ \text{split?}(x', s', s_1)\ \wedge\ \text{satisfies}(z', s_1)\ \wedge\ O(x', z')) \\ & \vdots \\ \hline \{1\} & ((\text{extract?}(z', s') \\ & \quad \vee\ (\exists\ (s :\ S) :\ (\text{transclosure?}(x', s', s)\ \wedge\ \text{extract?}(z', s)))) \\ & \quad \wedge\ O(x', z')) \end{array}$$

For the top quantifier in -1, we introduce Skolem constants: (s_1'),

Applying disjunctive simplification to flatten sequent,

Splitting conjunctions,

we get 2 subgoals:

`lemma2.2.2.1:`

$$\begin{array}{ll} \{\text{-}1\} & \text{split?}(x', s', s_1') \\ \{\text{-}2\} & \text{satisfies}(z', s_1') \\ \{\text{-}3\} & O(x', z') \\ & \vdots \\ \hline \{1\} & (\text{extract?}(z', s') \\ & \quad \vee\ (\exists\ (s :\ S) :\ (\text{transclosure?}(x', s', s)\ \wedge\ \text{extract?}(z', s)))) \end{array}$$

Applying satax where

Instantiating the top quantifier in -1 with the terms: $(x'\ s'\ s_1'\ z')$,

Applying disjunctive simplification to flatten sequent,

Invoking decision procedures,

Applying ax4 where

Instantiating the top quantifier in -1 with the terms: $(x'\ s'\ z')$,

Invoking decision procedures,

which is trivially true.

This completes the proof of `lemma2.2.2.1`.

`lemma2.2.2.2:`

$$\begin{array}{ll} \{\text{-}1\} & \text{split?}(x', s', s_1') \\ \{\text{-}2\} & \text{satisfies}(z', s_1') \\ \{\text{-}3\} & O(x', z') \\ & \vdots \\ \hline \{1\} & O(x', z') \end{array}$$

which is trivially true.

This completes the proof of `lemma2.2.2.2`.

Q.E.D.